TITLE: AUTHENTICATION OF RESOURCES USAGE IN A MULTI-USER ENVIRONMENT

FIELD OF INVENTION

The present invention relates to authenticated usage tracking of a resource shared by a number of consumers.

The invention has primarily been developed for use in printers to enable tracking of ink usage by multiple printer controllers and/or printheads. However, it will be appreciated that the invention can be applied to other fields in which analogous problems are faced.

BACKGROUND OF INVENTION

Manufacturing a printhead that has relatively high resolution and print-speed raises a number of problems.

Difficulties in manufacturing pagewidth printheads of any substantial size arise due to the relatively small dimensions of standard silicon wafers that are used in printhead (or printhead module) manufacture. For example, if it is desired to make an 8 inch wide pagewidth printhead, only one such printhead can be laid out on a standard 8-inch wafer, since such wafers are circular in plan. Manufacturing a pagewidth printhead from two or more smaller modules can reduce this limitation to some extent, but raises other problems related to providing a joint between adjacent printhead modules that is precise enough to avoid visible artefacts (which would typically take the form of noticeable lines) when the printhead is used. The problem is exacerbated in relatively high-resolution applications because of the tight tolerances dictated by the small spacing between nozzles.

The quality of a joint region between adjacent printhead modules relies on factors including a precision with which the abutting ends of each module can be manufactured, the accuracy with which they can be aligned when assembled into a single printhead, and other more practical factors such as management of ink channels behind the nozzles. It will be appreciated that the difficulties include relative vertical displacement of the printhead modules with respect to each other.

Whilst some of these issues may be dealt with by careful design and manufacture, the level of precision required renders it relatively expensive to manufacture printheads within the required tolerances. It would be desirable to provide a solution to one or more of the problems associated with precision manufacture and assembly of multiple printhead modules to form a printhead, and especially a pagewidth printhead.

In some cases, it is desirable to produce a number of different printhead module types or lengths on a substrate to maximise usage of the substrate's surface area. However, different sizes and types of modules will have different numbers and layouts of print nozzles, potentially including different horizontal and vertical offsets. Where two or more modules are to be joined to form a single printhead, there is also the problem of dealing with different seam shapes between abutting ends of joined modules, which again may incorporate vertical or horizontal offsets between the modules. Printhead controllers are usually dedicated application specific integrated circuits (ASICs) designed for specific use with a single type of printhead module, that is used by itself rather than with other modules. It would be desirable to provide a way in which different lengths and types of printhead modules could be accounted for using a single printer controller.

Printer controllers face other difficulties when two or more printhead modules are involved, especially if it is desired to send dot data to each of the printheads directly (rather than via a single printhead connected to the controller). One concern is that data delivered to different length controllers at the same rate will cause the shorter of the modules to be ready for printing before any longer modules. Where there is little difference involved, the issue may not be of importance, but for large length differences, the result is that the bandwidth of a shared memory from which the dot data is supplied to the modules is effectively left idle once one of the modules is full and the remaining module or modules is still being filled. It would be desirable to provide a way of improving memory bandwidth usage in a system comprising a plurality of printhead modules of uneven length.

In any printing system that includes multiple nozzles on a printhead or printhead module, there is the possibility of one or more of the nozzles failing in the field, or being inoperative due to manufacturing defect. Given the relatively large size of a typical printhead module, it would be desirable to provide some form of compensation for one or more " dead" nozzles. Where the printhead also outputs fixative on a per-nozzle basis, it is also desirable that the fixative is provided in such a way that dead nozzles are compensated for.

A printer controller can take the form of an integrated circuit, comprising a processor and one or more peripheral hardware units for implementing specific data manipulation functions. A number of these units and the processor may need access to a common resource such as memory. One way of arbitrating between multiple access requests for a common resource is timeslot arbitration, in which access to the resource is guaranteed to a particular requestor during a predetermined timeslot.

One difficulty with this arrangement lies in the fact that not all access requests make the same demands on the resource in terms of timing and latency. For example, a memory read requires that data be fetched from memory, which may take a number of cycles, whereas a memory write can commence immediately. Timeslot arbitration does not take into account these differences, which may result in accesses being performed in a less efficient manner than might otherwise be the case. It would be desirable to provide a timeslot arbitration scheme that improved this efficiency as compared with prior art timeslot arbitration

schemes.

Also of concern when allocating resources in a timeslot arbitration scheme is the fact that the priority of an access request may not be the same for all units. For example, it would be desirable to provide a timeslot arbitration scheme in which one requestor (typically the memory) is granted special priority such that its requests are dealt with earlier than would be the case in the absence of such priority.

In systems that use a memory and cache, a cache miss (in which an attempt to load data or an instruction from a cache fails) results in a memory access followed by a cache update. It is often desirable when updating the cache in this way to update data other than that which was actually missed. A typical example would be a cache miss for a byte resulting in an entire word or line of the cache associated with that byte being updated. However, this can have the effect of tying up bandwidth between the memory (or a memory manager) and the processor where the bandwidth is such that several cycles are required to transfer the entire word or line to the cache. It would be desirable to provide a mechanism for updating a cache that improved cache update speed and/or efficiency.

Most integrated circuits an externally provided signal as (or to generate) a clock, often provided from a dedicated clock generation circuit. This is often due to the difficulties of providing an onboard clock that can operate at a speed that is predictable. Manufacturing tolerances of such on-board clock generation circuitry can result in clock rates that vary by a factor of two, and operating temperatures can increase this margin by an additional factor of two. In some cases, the particular rate at which the clock operates is not of particular concern. However, where the integrated circuit will be writing to an internal circuit that is sensitive to the time over which a signal is provided, it may be undesirable to have the signal be applied for too long or short a time. For example, flash memory is sensitive to being written too for too long a period. It would be desirable to provide a mechanism for adjusting a rate of an on-chip system clock to take into account the impact of manufacturing variations on clockspeed.

One form of attacking a secure chip is to induce (usually by increasing) a clock speed that takes the logic outside its rated operating frequency. One way of doing this is to reduce the temperature of the integrated circuit, which can cause the clock to race. Above a certain frequency, some logic will start malfunctioning. In some cases, the malfunction can be such that information on the chip that would otherwise be secure may become available to an external connection. It would be desirable to protect an integrated circuit from such attacks.

In an integrated circuit comprising non-volatile memory, a power failure can result in unintentional behaviour. For example, if an address or data becomes unreliable due to falling voltage supplied to the circuit but there is still sufficient power to cause a write, incorrect data can be written. Even worse, the data (incorrect or not) could be written to the wrong memory. The problem is exacerbated with multi-word writes. It would be desirable to provide a mechanism for reducing or preventing spurious writes when power to an integrated circuit is failing.

In an integrated circuit, it is often desirable to reduce unauthorised access to the contents of memory. This is particularly the case where the memory includes a key or some other form of security information that allows the integrated circuit to communicate with another entity (such as another integrated circuit, for example) in a secure manner. It would be particularly advantageous to prevent attacks involving direct probing of memory addresses by physically investigating the chip (as distinct from electronic or logical attacks via manipulation of signals and power supplied to the integrated circuit).

It is also desirable to provide an environment where the manufacturer of the integrated circuit (or some other authorised entity) can verify or authorize code to be run on an integrated circuit.

Another desideratum would be the ability of two or more entities, such as integrated circuits, to communicate with each other in a secure manner. It would also be desirable to provide a mechanism for secure communication between a first entity and a second entity, where the two entities, whilst capable of some form of secure communication, are not able to establish such communication between themselves.

In a system that uses resources (such as a printer, which uses inks) it may be desirable to monitor and update a record related to resource usage. Authenticating ink quality can be a major issue, since the attributes of inks used by a given printhead can be quite specific. Use of incorrect ink can result in anything from misfiring or poor performance to damage or destruction of the printhead. It would therefore be desirable to provide a system that enables authentication of the correct ink being used, as well as providing various support systems secure enabling refilling of ink cartridges.

In a system that prevents unauthorized programs from being loaded onto or run on an integrated circuit, it can be laborious to allow developers of software to access the circuits during software development. Enabling access to integrated circuits of a particular type requires authenticating software with a relatively high-level key. Distributing the key for use by developers is inherently unsafe, since a single leak of the key outside the organization could endanger security of all chips that use a related key to authorize programs. Having a small number of people with high-security clearance available to authenticate programs for testing can be inconvenient, particularly in the case where frequent incremental changes in programs during development require testing. It would be desirable to provide a mechanism for allowing access to one or more integrated circuits without risking the security of other integrated circuits in a series of such integrated circuits.

In symmetric key security, a message, denoted by M, is *plaintext*. The process of transforming M into *ciphertext* C, where the substance of M is hidden, is called *encryption*. The process of transforming C back into M is called *decryption*. Referring to the encryption function as E, and the decryption function as D, we have the following identities:

$$E[M] = C$$
$$D[C] = M$$

Therefore the following identity is true:

$$D[E[M]] = M$$

5

A symmetric encryption algorithm is one where:
- the encryption function E relies on key K_1,
- the decryption function D relies on key K_2,
- K_2 can be derived from K_1, and

10
- K_1 can be derived from K_2.

In most symmetric algorithms, K_1 equals K_2. However, even if K_1 does not equal K_2, given that one key can be derived from the other, a single key K can suffice for the mathematical definition. Thus:

$$E_K[M] = C$$
$$D_K[C] = M$$

15

The security of these algorithms rests very much in the key K. Knowledge of K allows *anyone* to encrypt or decrypt. Consequently K must remain a secret for the duration of the value of M. For example, M may be a wartime message "My current position is grid position 123-456". Once the war is over the value of

20
M is greatly reduced, and if K is made public, the knowledge of the combat unit's position may be of no relevance whatsoever. The security of the particular symmetric algorithm is a function of two things: the strength of the algorithm and the length of the key.

An asymmetric encryption algorithm is one where:

25
- the encryption function E relies on key K_1,
- the decryption function D relies on key K_2,
- K_2 cannot be derived from K_1 in a reasonable amount of time, and
- K_1 cannot be derived from K_2 in a reasonable amount of time.

30
Thus:

$$E_{K1}[M] = C$$
$$D_{K2}[C] = M$$

These algorithms are also called *public-key* because one key K_1 can be made public. Thus anyone can encrypt a message (using K_1) but only the person with the corresponding decryption key (K_2) can decrypt

35
and thus read the message.

In most cases, the following identity also holds:

$$E_{K2}[M] = C$$
$$D_{K1}[C] = M$$

This identity is very important because it implies that anyone with the public key K_1 can see M and know that it came from the owner of K_2. No-one else could have generated C because to do so would imply knowledge of K_2. This gives rise to a different application, unrelated to encryption - digital signatures.

A number of public key cryptographic algorithms exist. Most are impractical to implement, and many generate a very large C for a given M or require enormous keys. Still others, while secure, are far too slow to be practical for several years. Because of this, many public key systems are hybrid - a public key mechanism is used to transmit a symmetric session key, and then the session key is used for the actual messages.

All of the algorithms have a problem in terms of key selection. A random number is simply not secure enough. The two large primes p and q must be chosen carefully - there are certain weak combinations that can be factored more easily (some of the weak keys can be tested for). But nonetheless, key selection is not a simple matter of randomly selecting 1024 bits for example. Consequently the key selection process must also be secure.

Symmetric and asymmetric schemes both suffer from a difficulty in allowing establishment of multiple relationships between one entity and a two or more others, without the need to provide multiple sets of keys. For example, if a main entity wants to establish secure communications with two or more additional entities, it will need to maintain a different key for each of the additional entities. For practical reasons, it is desirable to avoid generating and storing large numbers of keys. To reduce key numbers, two or more of the entities may use the same key to communicate with the main entity. However, this means that the main entity cannot be sure which of the entities it is communicating with. Similarly, messages from the main entity to one of the entities can be decrypted by any of the other entities with the same key. It would be desirable if a mechanism could be provided to allow secure communication between a main entity and one or more other entities that overcomes at least some of the shortcomings of prior art.

In a system where a first entity is capable of secure communication of some form, it may be desirable to establish a relationship with another entity without providing the other entity with any information related the first entity's security features. Typically, the security features might include a key or a cryptographic function. It would be desirable to provide a mechanism for enabling secure communications between a first and second entity when they do not share the requisite secret function, key or other relationship to enable them to establish trust.

A number of other aspects, features, preferences and embodiments are disclosed in the Detailed Description of the Preferred Embodiment below.

SUMMARY OF THE INVENTION

In accordance with a first embodiment of the invention, there is provided, in a system comprising a plurality of consumers of one or more common resources, a method of tracking usage of the one or more common resources, comprising the steps of:

5   from each consumer, broadcasting to each of the other consumers a value indicative of an amount of the one or more resources consumed;

at each of the consumers, receiving the broadcasted values from the other consumers; and

in each consumer, storing a record of the total of the values that the consumer broadcasted and the values received from the other consumers.

10

Preferably, a memory stores a total indicative of the sum of all values broadcast by the consumers, the method further comprising the steps of, for each of at least a plurality of the consumers:

performing an authenticated read of the total in the memory;

comparing the total in the consumer's record with the total read from the memory; and

15   in the event the totals do not match, performing an action.

Preferably, the memory is in one of the consumers and comprises that consumer's record.

Preferably, the action includes halting printing and or outputting an error message;

Preferably, the values are broadcast in a non-secure manner.

Preferably, the value is signless, thereby preventing recrediting of the total in the memory.

Preferably, the consumers are printer controllers.

Preferably, each of the printer controllers controls printing to a different part of print media to be printed.

Preferably, the resource is ink and the one or more values represent one or more corresponding inks consumed by one or more printheads associated with the printer controllers.

BRIEF DESCRIPTION OF THE DRAWINGS

Preferred and other embodiments of the invention will now be described, by way of example only, with reference to the accompanying drawings, in which:

Figure 1 is an example of state machine notation

Figure 2 shows document data flow in a printer

Figure 3 is an example of a single printer controller (hereinafter "SoPEC") A4 simplex printer system

Figure 4 is an example of a dual SoPEC A4 duplex printer system

Figure 5 is an example of a dual SoPEC A3 simplex printer system

Figure 6 is an example of a quad SoPEC A3 duplex printer system

Figure 7 is an example of a SoPEC A4 simplex printing system with an extra SoPEC used as DRAM storage

Figure 8 is an example of an A3 duplex printing system featuring four printing SoPECs

Figure 9 shows pages containing different numbers of bands

Figure 10 shows the contents of a page band

Figure 11 illustrates a page data path from host to SoPEC

Figure 12 shows a page structure

Figure 13 shows a SoPEC system top level partition

Figure 14 shows a SoPEC CPU memory map (not to scale)

Figure 15 is a block diagram of CPU

Figure 16 shows CPU bus transactions

Figure 17 shows a state machine for a CPU subsystem slave

Figure 18 shows a SoPEC CPU memory map (not to scale)

Figure 19 shows an external signal view of a memory management unit (hereinafter "MMU") sub-block partition

Figure 20 shows an internal signal view of an MMU sub-block partition

Figure 21 shows a DRAM write buffer

Figure 22 shows DIU waveforms for multiple transactions

Figure 23 shows a SoPEC LEON CPU core

Figure 24 shows a cache data RAM wrapper

Figure 25 shows a realtime debug unit block diagram

Figure 26 shows interrupt acknowledge cycles for single and pending interrupts

Figure 27 shows an A3 duplex system featuring four printing SoPECs with a single SoPEC DRAM device

Figure 28 is an SCB block diagram

Figure 29 is a logical view of the SCB of figure 28

Figure 30 shows an ISI configuration with four SoPEC devices

Figure 31 shows half-duplex interleaved transmission from ISIMaster to ISISlave

Figure 32 shows ISI transactions

Figure 33 shows an ISI long packet

Figure 34 shows an ISI ping packet

DETAILED DESCRIPTION OF PREFERRED AND OTHER EMBODIMENTS

It will be appreciated that the detailed description that follows takes the form of a highly detailed design of the invention, including supporting hardware and software. A high level of detailed disclosure is provided to ensure that one skilled in the art will have ample guidance for
5   implementing the invention.

Imperative phrases such as "must", "requires", "necessary" and "important" (and similar language) should be read as being indicative of being necessary only for the preferred embodiment actually being described. As such, unless the opposite is clear from the context, imperative wording should
10   not be interpreted as such. Nothing in the detailed description is to be understood as limiting the scope of the invention, which is intended to be defined as widely as is defined in the accompanying claims.

Indications of expected rates, frequencies, costs, and other quantitative values are exemplary and
15   estimated only, and are made in good faith. Nothing in this specification should be read as implying that a particular commercial embodiment is or will be capable of a particular performance level in any measurable area.

It will be appreciated that the principles, methods and hardware described throughout this document
20   can be applied to other fields. Much of the security-related disclosure, for example, can be applied to many other fields that require secure communications between entities, and certainly has application far beyond the field of printers.

SYSTEM OVERVIEW
25   The preferred of the present invention is implemented in a printer using microelectromechanical systems (MEMS) printheads. The printer can receive data from, for example, a personal computer such as an IBM compatible PC or Apple computer. In other embodiments, the printer can receive data directly from, for example, a digital still or video camera. The particular choice of communication link is not important, and can be based, for example, on USB, Firewire, Bluetooth or
30   any other wireless or hardwired communications protocol.

PRINT SYSTEM OVERVIEW
3   Introduction
This document describes the SoPEC (Small office home office Print Engine Controller) ASIC
35   (Application Specific Integrated Circuit) suitable for use in, for example, SoHo printer products. The SoPEC ASIC is intended to be a low cost solution for bi-lithic printhead control, replacing the multichip solutions in larger more professional systems with a single chip. The increased cost competitiveness is achieved by integrating several systems such as a modified PEC1 printing pipeline, CPU control system, peripherals and memory sub-system onto one SoC ASIC, reducing
40   component count and simplifying board design.

This section will give a general introduction to Memjet printing systems, introduce the components that make a bi-lithic printhead system, describe possible system architectures and show how several SoPECs can be used to achieve A3 and A4 duplex printing. The section "SoPEC ASIC" describes the SoC SoPEC ASIC, with subsections describing the CPU, DRAM and Print Engine Pipeline subsystems. Each section gives a detailed description of the blocks used and their operation within the overall print system. The final section describes the bi-lithic printhead construction and associated implications to the system due to its makeup.

4 Nomenclature

4.1 BI-LITHIC PRINTHEAD NOTATION

A bi-lithic based printhead is constructed from 2 printhead ICs of varying sizes. The notation M:N is used to express the size relationship of each IC, where M specifies one printhead IC in inches and N specifies the remaining printhead IC in inches.

The 'SoPEC/MoPEC Bilithic Printhead Reference' document [10] contains a description of the bi-lithic printhead and related terminology.

4.2 DEFINITIONS

The following terms are used throughout this specification:

Bi-lithic printhead	Refers to printhead constructed from 2 printhead ICs
CPU	Refers to CPU core, caching system and MMU.
ISI-Bridge chip	A device with a high speed interface (such as USB2.0, Ethernet or IEEE1394) and one or more ISI interfaces. The ISI-Bridge would be the ISIMaster for each of the ISI buses it interfaces to.
ISIMaster	The ISIMaster is the only device allowed to initiate communication on the Inter Sopec Interface (ISI) bus. The ISIMaster interfaces with the host.
ISISlave	Multi-SoPEC systems will contain one or more ISISlave SoPECs connected to the ISI bus. ISISlaves can only respond to communication initiated by the ISIMaster.
LEON	Refers to the LEON CPU core.
LineSyncMaster	The LineSyncMaster device generates the line synchronisation pulse that all SoPECs in the system must synchronise their line outputs to.
Multi-SoPEC	Refers to SoPEC based print system with multiple SoPEC devices
Netpage	Refers to page printed with tags (normally in infrared ink).
PEC1	Refers to Print Engine Controller version 1, precursor to SoPEC used to control printheads constructed from multiple angled printhead segments.
Printhead IC	Single MEMS IC used to construct bi-lithic printhead
PrintMaster	The PrintMaster device is responsible for coordinating all aspects of the print operation. There may only be one PrintMaster in a system.

QA Chip	Quality Assurance Chip
Storage SoPEC	An ISISlave SoPEC used as a DRAM store and which does not print.
Tag	Refers to pattern which encodes information about its position and orientation which allow it to be optically located and its data contents read.

5   4.3   ACRONYM AND ABBREVIATIONS

The following acronyms and abbreviations are used in this specification

CFU	Contone FIFO Unit
CPU	Central Processing Unit
DIU	DRAM Interface Unit
DNC	Dead Nozzle Compensator
DRAM	Dynamic Random Access Memory
DWU	DotLine Writer Unit
GPIO	General Purpose Input Output
HCU	Halftoner Compositor Unit
ICU	Interrupt Controller Unit
ISI	Inter SoPEC Interface
LDB	Lossless Bi-level Decoder
LLU	Line Loader Unit
LSS	Low Speed Serial interface
MEMS	Micro Electro Mechanical System
MMU	Memory Management Unit
PCU	SoPEC Controller Unit
PHI	PrintHead Interface
PSS	Power Save Storage Unit
RDU	Real-time Debug Unit
ROM	Read Only Memory
SCB	Serial Communication Block
SFU	Spot FIFO Unit
SMG4	Silverbrook Modified Group 4.
SoPEC	Small office home office Print Engine Controller
SRAM	Static Random Access Memory
TE	Tag Encoder
TFU	Tag FIFO Unit
TIM	Timers Unit
USB	Universal Serial Bus

4.4   PSEUDOCODE NOTATION

In general the pseudocode examples use C like statements with some exceptions.

Symbol and naming convections used for pseudocode.

//	Comment
=	Assignment

==,!=,<,>	Operator equal, not equal, less than, greater than	
+,-,*,/,%	Operator addition, subtraction, multiply, divide, modulus	
&,|,^,<<,>>,~	Bitwise AND, bitwise OR, bitwise exclusive OR, left shift, right shift, complement	
AND,OR,NOT	Logical AND, Logical OR, Logical inversion	
[XX:YY]	Array/vector specifier	
{a, b, c}	Concatenation operation	
++, --	Increment and decrement	

4.4.1 Register and signal naming conventions

In general register naming uses the C style conventions with capitalization to denote word delimiters. Signals use RTL style notation where underscore denote word delimiters. There is a direct translation between both convention. For example the *CmdSourceFifo* register is equivalent to *cmd_source_fifo* signal.

4.5 STATE MACHINE NOTATION

State machines should be described using the pseudocode notation outlined above. State machine descriptions use the convention of <u>underline</u> to indicate the cause of a transition from one state to another and plain text (no underline) to indicate the effect of the transition i.e. signal transitions which occur when the new state is entered.

A sample state machine is shown in Figure 1.

5 Printing Considerations

A bi-lithic printhead produces 1600 dpi bi-level dots. On low-diffusion paper, each ejected drop forms a 22.5μm diameter dot. Dots are easily produced in isolation, allowing dispersed-dot dithering to be exploited to its fullest. Since the bi-lithic printhead is the width of the page and operates with a constant paper velocity, color planes are printed in perfect registration, allowing ideal dot-on-dot printing. Dot-on-dot printing minimizes 'muddying' of midtones caused by inter-color bleed.

A page layout may contain a mixture of images, graphics and text. Continuous-tone (contone) images and graphics are reproduced using a stochastic dispersed-dot dither. Unlike a clustered-dot (or amplitude-modulated) dither, a *dispersed-dot* (or frequency-modulated) dither reproduces high spatial frequencies (i.e. image detail) almost to the limits of the dot resolution, while simultaneously reproducing lower spatial frequencies to their full color depth, when spatially integrated by the eye.

A *stochastic* dither matrix is carefully designed to be free of objectionable low-frequency patterns when tiled across the image. As such its size typically exceeds the minimum size required to support a particular number of intensity levels (e.g. 16×16× 8 bits for 257 intensity levels).

Human contrast sensitivity peaks at a spatial frequency of about 3 cycles per degree of visual field and then falls off logarithmically, decreasing by a factor of 100 beyond about 40 cycles per degree and becoming immeasurable beyond 60 cycles per degree [25][25]. At a normal viewing distance of 12 inches (about 300mm), this translates roughly to 200-300 cycles per inch (cpi) on the printed page, or 400-600 samples per inch according to Nyquist's theorem.

In practice, contone resolution above about 300 ppi is of limited utility outside special applications such as medical imaging. Offset printing of magazines, for example, uses contone resolutions in the range 150 to 300 ppi. Higher resolutions contribute slightly to color error through the dither.

Black text and graphics are reproduced directly using bi-level black dots, and are therefore not anti-aliased (i.e. low-pass filtered) before being printed. Text should therefore be *supersampled* beyond the perceptual limits discussed above, to produce smoother edges when spatially integrated by the eye. Text resolution up to about 1200 dpi continues to contribute to perceived text sharpness (assuming low-diffusion paper, of course).

A Netpage printer, for example, may use a contone resolution of 267 ppi (i.e. 1600 dpi / 6), and a black text and graphics resolution of 800 dpi. A high end office or departmental printer may use a contone resolution of 320 ppi (1600 dpi / 5) and a black text and graphics resolution of 1600 dpi. Both formats are capable of exceeding the quality of commercial (offset) printing and photographic reproduction.

.6   Document Data Flow

6.1   CONSIDERATIONS

Because of the page-width nature of the bi-lithic printhead, each page must be printed at a constant speed to avoid creating visible artifacts. This means that the printing speed can't be varied to match the input data rate. Document rasterization and document printing are therefore decoupled to ensure the printhead has a constant supply of data. A page is never printed until it is fully rasterized. This can be achieved by storing a compressed version of each rasterized page image in memory. This decoupling also allows the RIP(s) to run ahead of the printer when rasterizing simple pages, buying time to rasterize more complex pages.

Because contone color images are reproduced by stochastic dithering, but black text and line graphics are reproduced directly using dots, the compressed page image format contains a separate foreground bi-level black layer and background contone color layer. The black layer is composited over the contone layer after the contone layer is dithered (although the contone layer has an optional black component). A final layer of Netpage tags (in infrared or black ink) is optionally added to the page for printout.

Figure 2 shows the flow of a document from computer system to printed page.

At 267 ppi for example, a A4 page (8.26 inches × 11.7 inches) of contone CMYK data has a size of 26.3MB. At 320 ppi, an A4 page of contone data has a size of 37.8MB. Using lossy contone compression algorithms such as JPEG [27], contone images compress with a ratio up to 10:1 without noticeable loss of quality, giving compressed page sizes of 2.63MB at 267 ppi and 3.78 MB at 320 ppi.

At 800 dpi, a A4 page of bi-level data has a size of 7.4MB. At 1600 dpi, a Letter page of bi-level data has a size of 29.5 MB. Coherent data such as text compresses very well. Using lossless bi-level compression algorithms such as SMG4 fax as discussed in Section 8.1.2.3.1, ten-point plain text compresses with a ratio of about 50:1. Lossless bi-level compression across an average page is about 20:1 with 10:1 possible for pages which compress poorly. The requirement for SoPEC is to be able to print text at 10:1 compression. Assuming 10:1 compression gives compressed page sizes of 0.74 MB at 800 dpi, and 2.95 MB at 1600 dpi.

Once dithered, a page of CMYK contone image data consists of 116MB of bi-level data. Using lossless bi-level compression algorithms on this data is pointless precisely because the optimal dither is stochastic - i.e. since it introduces hard-to-compress disorder.

Netpage tag data is optionally supplied with the page image. Rather than storing a compressed bi-level data layer for the Netpage tags, the tag data is stored in its raw form. Each tag is supplied up to 120 bits of raw variable data (combined with up to 56 bits of raw fixed data) and covers up to a 6mm × 6mm area (at 1600 dpi). The absolute maximum number of tags on a A4 page is 15,540 when the tag is only 2mm × 2mm (each tag is 126 dots × 126 dots, for a total coverage of 148 tags × 105 tags). 15,540 tags of 128 bits per tag gives a compressed tag page size of 0.24 MB.

The multi-layer compressed page image format therefore exploits the relative strengths of lossy JPEG contone image compression, lossless bi-level text compression, and tag encoding. The format is compact enough to be storage-efficient, and simple enough to allow straightforward real-time expansion during printing.

Since text and images normally don't overlap, the normal worst-case page image size is image only, while the normal best-case page image size is text only. The addition of worst case Netpage tags adds 0.24MB to the page image size. The worst-case page image size is text over image plus tags. The average page size assumes a quarter of an average page contains images. Table 1 shows data sizes for compressed Letter page for these different options.

Table 1. Data sizes for A4 page (8.26 inches × 11.7 inches)

	267 ppi contone 800 dpi bi-level	320 ppi contone 1600 dpi bi-level
Image only (contone), 10:1 compression	2.63 MB	3.78 MB
Text only (bi-level), 10:1 compression	0.74 MB	2.95 MB
Netpage tags, 1600 dpi	0.24 MB	0.24 MB
Worst case (text + image + tags)	3.61 MB	6.67 MB
Average (text + 25% image + tags)	1.64 MB	4.25 MB

6.2 DOCUMENT DATA FLOW

The Host PC rasterizes and compresses the incoming document on a page by page basis. The page is restructured into bands with one or more bands used to construct a page. The compressed data is then transferred to the SoPEC device via the USB link. A complete band is stored in SoPEC embedded memory. Once the band transfer is complete the SoPEC device reads the compressed data, expands the band, normalizes contone, bi-level and tag data to 1600 dpi and transfers the resultant calculated dots to the bi-lithic printhead.

The document data flow is

- The RIP software rasterizes each page description and compress the rasterized page image.
- The infrared layer of the printed page optionally contains encoded Netpage [5] tags at a programmable density.

- The compressed page image is transferred to the SoPEC device via the USB normally on a band by band basis.
- The print engine takes the compressed page image and starts the page expansion.
- The first stage page expansion consists of 3 operations performed in parallel
- expansion of the JPEG-compressed contone layer
- expansion of the SMG4 fax compressed bi-level layer
- encoding and rendering of the bi-level tag data.
- The second stage dithers the contone layer using a programmable dither matrix, producing up to four bi-level layers at full-resolution.
- The second stage then composites the bi-level tag data layer, the bi-level SMG4 fax de-compressed layer and up to four bi-level JPEG de-compressed layers into the full-resolution page image.
- A fixative layer is also generated as required.
- The last stage formats and prints the bi-level data through the bi-lithic printhead via the printhead interface.

The SoPEC device can print a full resolution page with 6 color planes. Each of the color planes can be generated from compressed data through any channel (either JPEG compressed, bi-level SMG4 fax compressed, tag data generated, or fixative channel created) with a maximum number of 6 data channels from page RIP to bi-lithic printhead color planes.

The mapping of data channels to color planes is programmable, this allows for multiple color planes in the printhead to map to the same data channel to provide for redundancy in the printhead to assist dead nozzle compensation.

Also a data channel could be used to gate data from another data channel. For example in stencil mode, data from the bilevel data channel at 1600 dpi can be used to filter the contone data channel at 320 dpi, giving the effect of 1600 dpi contone image.

6.3 PAGE CONSIDERATIONS DUE TO SoPEC

The SoPEC device typically stores a complete page of document data on chip. The amount of storage available for compressed pages is limited to 2Mbytes, imposing a fixed maximum on compressed page size. A comparison of the compressed image sizes in Table 2 indicates that SoPEC would not be capable of printing worst case pages unless they are split into bands and printing commences before all the bands for the page have been downloaded. The page sizes in the table are shown for comparison purposes and would be considered reasonable for a professional level printing system. The SoPEC device is aimed at the consumer level and would not be required to print pages of that complexity. Target document types for the SoPEC device are shown Table 2.

Table 2. Page content targets for SoPEC

Page Content Description	Calculation	Size (MByte)

Best Case picture Image, 267ppi with 3 colors, A4 size	8.26x11.7x267x267x3 @10:1	1.97
Full page text, 800dpi A4 size	8.26x11.7x800x800 @10:1	0.74
Mixed Graphics and Text - Image of 6 inches x 4 inches @ 267 ppi and 3 colors - Remaining area text ~73 inches^2, 800 dpi	6x4x267x267x3 @ 5:1 800x800x73 @ 10:1	1.55
Best Case Photo, 3 Colors, 6.6 MegaPixel Image	6.6 Mpixel @ 10:1	2.00

If a document with more complex pages is required, the page RIP software in the host PC can determine that there is insufficient memory storage in the SoPEC for that document. In such cases the RIP software can take two courses of action. It can increase the compression ratio until the compressed page size will fit in the SoPEC device, at the expense of document quality, or divide the page into bands and allow SoPEC to begin printing a page band before all bands for that page are downloaded. Once SoPEC starts printing a page it cannot stop, if SoPEC consumes compressed data faster than the bands can be downloaded a buffer underrun error could occur causing the print to fail. A buffer underrun occurs if a line synchronisation pulse is received before a line of data has been transferred to the printhead.

Other options which can be considered if the page does not fit completely into the compressed page store are to slow the printing or to use multiple SoPECs to print parts of the page. A Storage SoPEC (Section 7.2.5) could be added to the system to provide guaranteed bandwidth data delivery. The print system could also be constructed using an ISI-Bridge chip (Section 7.2.6) to provide guaranteed data delivery.

7   Memjet Printer Architecture

The SoPEC device can be used in several printer configurations and architectures.

In the general sense every SoPEC based printer architecture will contain:

- One or more SoPEC devices.
- One or more bi-lithic printheads.
- Two or more LSS busses.
- Two or more QA chips.
- USB 1.1 connection to host or ISI connection to Bridge Chip.
- ISI bus connection between SoPECs (when multiple SoPECs are used).

Some example printer configurations as outlined in Section 7.2. The various system components are outlined briefly in Section 7.1.

7.1   SYSTEM COMPONENTS

7.1.1 SoPEC Print Engine Controller

The SoPEC device contains several system on a chip (SoC) components, as well as the print engine pipeline control application specific logic.

7.1.1.1   Print Engine Pipeline (PEP) Logic

The PEP reads compressed page store data from the embedded memory, optionally decompresses the data and formats it for sending to the printhead. The print engine pipeline functionality includes expanding the page image, dithering the contone layer, compositing the black layer over the contone layer, rendering of Netpage tags, compensation for dead nozzles in the printhead, and

5   sending the resultant image to the bi-lithic printhead.

7.1.1.2 *Embedded CPU*

SoPEC contains an embedded CPU for general purpose system configuration and management. The CPU performs page and band header processing, motor control and sensor monitoring (via the GPIO) and other system control functions. The CPU can perform buffer management or report

10   buffer status to the host. The CPU can optionally run vendor application specific code for general print control such as paper ready monitoring and LED status update.

7.1.1.3 *Embedded Memory Buffer*

A 2.5Mbyte embedded memory buffer is integrated onto the SoPEC device, of which approximately 2Mbytes are available for compressed page store data. A compressed page is divided into one or

15   more bands, with a number of bands stored in memory. As a band of the page is consumed by the PEP for printing a new band can be downloaded. The new band may be for the current page or the next page.

Using banding it is possible to begin printing a page before the complete compressed page is downloaded, but care must be taken to ensure that data is always available for printing or a buffer

20   underrun may occur.

An Storage SoPEC acting as a memory buffer (Section 7.2.5) or an ISI-Bridge chip with attached DRAM (Section 7.2.6) could be used to provide guaranteed data delivery.

7.1.1.4 *Embedded USB 1.1 Device*

The embedded USB 1.1 device accepts compressed page data and control commands from the

25   host PC, and facilitates the data transfer to either embedded memory or to another SoPEC device in multi-SoPEC systems.

7.1.2   Bi-lithic Printhead

The printhead is constructed by abutting 2 printhead ICs together. The printhead ICs can vary in size from 2 inches to 8 inches, so to produce an A4 printhead several combinations are possible.

30   For example two printhead ICs of 7 inches and 3 inches could be used to create a A4 printhead (the notation is 7:3). Similarly 6 and 4 combination (6:4), or 5:5 combination. For an A3 printhead it can be constructed from 8:6 or an 7:7 printhead IC combination. For photographic printing smaller printheads can be constructed.

7.1.3   LSS interface bus

35   Each SoPEC device has 2 LSS system buses for communication with QA devices for system authentication and ink usage accounting. The number of QA devices per bus and their position in the system is unrestricted with the exception that *PRINTER_QA* and *INK_QA* devices should be on separate LSS busses.

7.1.4   QA devices

Each SoPEC system can have several QA devices. Normally each printing SoPEC will have an associated *PRINTER_QA*. Ink cartridges will contain an *INK_QA* chip. *PRINTER_QA* and *INK_QA* devices should be on separate LSS busses. All QA chips in the system are physically identical with flash memory contents defining *PRINTER_QA* from *INK_QA* chip.

7.1.5   ISI interface

The Inter-SoPEC Interface (ISI) provides a communication channel between SoPECs in a multi-SoPEC system. The ISIMaster can be SoPEC device or an ISI-Bridge chip depending on the printer configuration. Both compressed data and control commands are transferred via the interface.

7.1.6   ISI-Bridge Chip

A device, other than a SoPEC with a USB connection, which provides print data to a number of slave SoPECs. A bridge chip will typically have a high bandwidth connection, such as USB2.0, Ethernet or IEEE1394, to a host and may have an attached external DRAM for compressed page storage. A bridge chip would have one or more ISI interfaces. The use of multiple ISI buses would allow the construction of independent print systems within the one printer. The ISI-Bridge would be the ISIMaster for each of the ISI buses it interfaces to.

7.2   POSSIBLE SoPEC SYSTEMS

Several possible SoPEC based system architectures exist. The following sections outline some possible architectures. It is possible to have extra SoPEC devices in the system used for DRAM storage. The QA chip configurations shown are indicative of the flexibility of LSS bus architecture, but not limited to those configurations.

7.2.1   A4 Simplex with 1 SoPEC device

In Figure 3, a single SoPEC device can be used to control two printhead ICs. The SoPEC receives compressed data through the USB device from the host. The compressed data is processed and transferred to the printhead.

7.2.2   A4 Duplex with 2 SoPEC devices

In Figure 4, two SoPEC devices are used to control two bi-lithic printheads, each with two printhead ICs. Each bi-lithic printhead prints to opposite sides of the same page to achieve duplex printing. The SoPEC connected to the USB is the ISIMaster SoPEC, the remaining SoPEC is an ISISlave. The ISIMaster receives all the compressed page data for both SoPECs and re-distributes the compressed data over the Inter-SoPEC Interface (ISI) bus.

It may not be possible to print an A4 page every 2 seconds in this configuration since the USB 1.1 connection to the host may not have enough bandwidth. An alternative would be for each SoPEC to have its own USB 1.1 connection. This would allow a faster average print speed.

7.2.3   A3 Simplex with 2 SoPEC devices

In Figure 5, two SoPEC devices are used to control one A3 bi-lithic printhead. Each SoPEC controls only one printhead IC (the remaining PHI port typically remains idle). This system uses the SoPEC with the USB connection as the ISIMaster. In this dual SoPEC configuration the compressed page store data is split across 2 SoPECs giving a total of 4Mbyte page store, this allows the system to

use compression rates as in an A4 architecture, but with the increased page size of A3. The ISIMaster receives all the compressed page data for all SoPECs and re-distributes the compressed data over the Inter-SoPEC Interface (ISI) bus.

It may not be possible to print an A3 page every 2 seconds in this configuration since the USB 1.1 connection to the host will only have enough bandwidth to supply 2Mbytes every 2 seconds. Pages which require more than 2MBytes every 2 seconds will therefore print more slowly. An alternative would be for each SoPEC to have its own USB 1.1 connection. This would allow a faster average print speed.

7.2.4   A3 Duplex with 4 SoPEC devices

In Figure 6 a 4 SoPEC system is shown. It contains 2 A3 bi-lithic printheads, one for each side of an A3 page. Each printhead contain 2 printhead ICs, each printhead IC is controlled by an independent SoPEC device, with the remaining PHI port typically unused. Again the SoPEC with USB 1.1 connection is the ISIMaster with the other SoPECs as ISISlaves. In total, the system contains 8Mbytes of compressed page store (2Mbytes per SoPEC), so the increased page size does not degrade the system print quality, from that of an A4 simplex printer. The ISIMaster receives all the compressed page data for all SoPECs and re-distributes the compressed data over the Inter-SoPEC Interface (ISI) bus.

It may not be possible to print an A3 page every 2 seconds in this configuration since the USB 1.1 connection to the host will only have enough bandwidth to supply 2Mbytes every 2 seconds. Pages which require more than 2MBytes every 2 seconds will therefore print more slowly. An alternative would be for each SoPEC or set of SoPECs on the same side of the page to have their own USB 1.1 connection (as ISISlaves may also have direct USB connections to the host). This would allow a faster average print speed.

7.2.5   SoPEC DRAM storage solution: A4 Simplex with 1 printing SoPEC and 1 memory SoPEC

Extra SoPECs can be used for DRAM storage e.g. in Figure 7 an A4 simplex printer can be built with a single extra SoPEC used for DRAM storage. The DRAM SoPEC can provide guaranteed bandwidth delivery of data to the printing SoPEC. SoPEC configurations can have multiple extra SoPECs used for DRAM storage.

7.2.6   ISI-Bridge chip solution: A3 Duplex system with 4 SoPEC devices

In Figure 8, an ISI-Bridge chip provides slave-only ISI connections to SoPEC devices. Figure 8 shows a ISI-Bridge chip with 2 separate ISI ports. The ISI-Bridge chip is the ISIMaster on each of the ISI busses it is connected to. All connected SoPECs are ISISlaves. The ISI-Bridge chip will typically have a high bandwidth connection to a host and may have an attached external DRAM for compressed page storage.

An alternative to having a ISI-Bridge chip would be for each SoPEC or each set of SoPECs on the same side of a page to have their own USB 1.1 connection. This would allow a faster average print speed.

8   Page Format and Printflow

When rendering a page, the RIP produces a page header and a number of bands (a non-blank page requires at least one band) for a page. The page header contains high level rendering

30

parameters, and each band contains compressed page data. The size of the band will depend on the memory available to the RIP, the speed of the RIP, and the amount of memory remaining in SoPEC while printing the previous band(s). Figure 9 shows the high level data structure of a number of pages with different numbers of bands in the page.

5   Each compressed band contains a mandatory band header, an optional bi-level plane, optional sets of interleaved contone planes, and an optional tag data plane (for Netpage enabled applications). Since each of these planes is optional[1], the band header specifies which planes are included with the band. Figure 10 gives a high-level breakdown of the contents of a page band.

10   A single SoPEC has maximum rendering restrictions as follows:
     • 1 bi-level plane
     • 1 contone interleaved plane set containing a maximum of 4 contone planes
     • 1 tag data plane
     • a bi-lithic printhead with a maximum of 2 printhead ICs
15   The requirement for single-sided A4 single SoPEC printing is
     • average contone JPEG compression ratio of 10:1, with a local minimum compression ratio of 5:1 for a single line of interleaved JPEG blocks.
     • average bi-level compression ratio of 10:1, with a local minimum compression ratio of 1:1 for a single line.
20   If the page contains rendering parameters that exceed these specifications, then the RIP or the Host PC must split the page into a format that can be handled by a single SoPEC.
     In the general case, the SoPEC CPU must analyze the page and band headers and generate an appropriate set of register write commands to configure the units in SoPEC for that page. The various bands are passed to the destination SoPEC(s) to locations in DRAM determined by the
25   host.
     The host keeps a memory map for the DRAM, and ensures that as a band is passed to a SoPEC, it is stored in a suitable free area in DRAM. Each SoPEC is connected to the ISI bus or USB bus via its Serial communication Block (SCB). The SoPEC CPU configures the SCB to allow compressed data bands to pass from the USB or ISI through the SCB to SoPEC DRAM. Figure 11 shows an
30   example data flow for a page destined to be printed by a single SoPEC. Band usage information is generated by the individual SoPECs and passed back to the host.

     SoPEC has an addressing mechanism that permits circular band memory allocation, thus facilitating easy memory management. However it is not strictly necessary that all bands be stored together.
35   As long as the appropriate registers in SoPEC are set up for each band, and a given band is contiguous[2], the memory can be allocated in any way.

---

[1]Although a band must contain at least one plane

[2]Contiguous allocation also includes wrapping around in SoPEC's band store memory.

8.1   PRINT ENGINE EXAMPLE PAGE FORMAT

This section describes a possible format of compressed pages expected by the embedded CPU in SoPEC. The format is generated by software in the host PC and interpreted by embedded software in SoPEC. This section indicates the type of information in a page format structure, but
5   implementations need not be limited to this format. The host PC can optionally perform the majority of the header processing.

The compressed format and the print engines are designed to allow real-time page expansion during printing, to ensure that printing is never interrupted in the middle of a page due to data underrun.
10   The page format described here is for a single black bi-level layer, a contone layer, and a Netpage tag layer. The black bi-level layer is defined to composite *over* the contone layer.

The black bi-level layer consists of a bitmap containing a 1-bit *opacity* for each pixel. This black layer *matte* has a resolution which is an integer or non-integer factor of the printer's dot resolution. The highest supported resolution is 1600 dpi, i.e. the printer's full dot resolution.
15   The contone layer, optionally passed in as YCrCb, consists of a 24-bit CMY or 32-bit CMYK *color* for each pixel. This contone image has a resolution which is an integer or non-integer factor of the printer's dot resolution. The requirement for a single SoPEC is to support 1 side per 2 seconds A4/Letter printing at a resolution of 267 ppi, i.e. one-sixth the printer's dot resolution.

Non-integer scaling can be performed on both the contone and bi-level images. Only integer
20   scaling can be performed on the tag data.

The black bi-level layer and the contone layer are both in compressed form for efficient storage in the printer's internal memory.

8.1.1   Page structure

A single SoPEC is able to print with full edge bleed for Letter and A3 via different stitch part
25   combinations of the bi-lithic printhead. It imposes no margins and so has a printable page area which corresponds to the size of its paper. The target page size is constrained by the printable page area, less the explicit (target) left and top margins specified in the page description. These relationships are illustrated below.

8.1.2   Compressed page format
30   Apart from being implicitly defined in relation to the printable page area, each page description is complete and self-contained. There is no data stored separately from the page description to which the page description refers.[3] The page description consists of a page header which describes the size and resolution of the page, followed by one or more page bands which describe the actual page content.

8.1.2.1   *Page header*
35   Table 3 shows an example format of a page header.

---

[3]SoPEC relies on dither matrices and tag structures to have already been set up, but these are not considered to be part of a general page format. It is trivial to extend the page format to allow exact specification of dither matrices and tag structures.

Table 3. Page header format

field	format	description
signature	16-bit integer	Page header format signature.
version	16-bit integer	Page header format version number.
structure size	16-bit integer	Size of page header.
band count	16-bit integer	Number of bands specified for this page.
target resolution (dpi)	16-bit integer	Resolution of target page. This is always 1600 for the Memjet printer.
target page width	16-bit integer	Width of target page, in dots.
target page height	32-bit integer	Height of target page, in dots.
target left margin for black and contone	16-bit integer	Width of target left margin, in dots, for black and contone.
target top margin for black and contone	16-bit integer	Height of target top margin, in dots, for black and contone.
target right margin for black and contone	16-bit integer	Width of target right margin, in dots, for black and contone.
target bottom margin for black and contone	16-bit integer	Height of target bottom margin, in dots, for black and contone.
target left margin for tags	16-bit integer	Width of target left margin, in dots, for tags.
target top margin for tags	16-bit integer	Height of target top margin, in dots, for tags.
target right margin for tags	16-bit integer	Width of target right margin, in dots, for tags.
target bottom margin for tags	16-bit integer	Height of target bottom margin, in dots, for tags.
generate tags	16-bit integer	Specifies whether to generate tags for this page (0 - no, 1 - yes).
fixed tag data	128-bit integer	This is only valid if generate tags is set.
tag vertical scale factor	16-bit integer	Scale factor in vertical direction from tag data resolution to target resolution. Valid range = 1-511. Integer scaling only
tag horizontal scale factor	16-bit integer	Scale factor in horizontal direction from tag data resolution to target resolution. Valid range = 1-511. Integer scaling only.
bi-level layer vertical scale factor	16-bit integer	Scale factor in vertical direction from bi-level resolution to target resolution (must be 1 or greater). May be non-integer. Expressed as a fraction with upper 8-bits the numerator and the lower 8 bits the denominator.

bi-level layer horizontal scale factor	16-bit integer	Scale factor in horizontal direction from bi-level resolution to target resolution (must be 1 or greater). May be non-integer. Expressed as a fraction with upper 8-bits the numerator and the lower 8 bits the denominator.
bi-level layer page width	16-bit integer	Width of bi-level layer page, in pixels.
bi-level layer page height	32-bit integer	Height of bi-level layer page, in pixels.
contone flags	16 bit integer	Defines the color conversion that is required for the JPEG data. Bits 2-0 specify how many contone planes there are (e.g. 3 for CMY and 4 for CMYK). Bit 3 specifies whether the first 3 color planes need to be converted back from YCrCb to CMY. Only valid if b2-0 = 3 or 4. 0 - no conversion, leave JPEG colors alone 1 - color convert. Bits 7-4 specifies whether the YCrCb was generated directly from CMY, or whether it was converted to RGB first via the step: R = 255-C, G = 255-M, B = 255-Y. Each of the color planes can be individually inverted. Bit 4: 0 - do not invert color plane 0 1 - invert color plane 0 Bit 5: 0 - do not invert color plane 1 1 - invert color plane 1 Bit 6: 0 - do not invert color plane 2 1 - invert color plane 2 Bit 7: 0 - do not invert color plane 3 1 - invert color plane 3 Bit 8 specifies whether the contone data is JPEG compressed or non-compressed: 0 - JPEG compressed 1 - non-compressed The remaining bits are reserved (0).
contone vertical scale factor	16-bit integer	Scale factor in vertical direction from contone channel resolution to target resolution. Valid

		range = 1-255. May be non-integer. Expressed as a fraction with upper 8-bits the numerator and the lower 8 bits the denominator.
contone horizontal scale factor	16-bit integer	Scale factor in horizontal direction from contone channel resolution to target resolution. Valid range = 1-255. May be non-integer. Expressed as a fraction with upper 8-bits the numerator and the lower 8 bits the denominator.
contone page width	16-bit integer	Width of contone page, in contone pixels.
contone page height	32-bit integer	Height of contone page, in contone pixels.
reserved	up to 128 bytes	Reserved and 0 pads out page header to multiple of 128 bytes.

The page header contains a signature and version which allow the CPU to identify the page header format. If the signature and/or version are missing or incompatible with the CPU, then the CPU can reject the page.

The contone flags define how many contone layers are present, which typically is used for defining whether the contone layer is CMY or CMYK. Additionally, if the color planes are CMY, they can be optionally stored as YCrCb, and further optionally color space converted from CMY directly or via RGB. Finally the contone data is specified as being either JPEG compressed or non-compressed.

The page header defines the resolution and size of the target page. The bi-level and contone layers are clipped to the target page if necessary. This happens whenever the bi-level or contone scale factors are not factors of the target page width or height.

The target left, top, right and bottom margins define the positioning of the target page within the printable page area.

The tag parameters specify whether or not Netpage tags should be produced for this page and what orientation the tags should be produced at (landscape or portrait mode). The fixed tag data is also provided.

The contone, bi-level and tag layer parameters define the page size and the scale factors.

8.1.2.2 Band format

Table 4 shows the format of the page band header.

Table 4. Band header format

field	format	description
signature	16-bit integer	Page band header format signature.

version	16-bit integer	Page band header format version number.
structure size	16-bit integer	Size of page band header.
bi-level layer band height	16-bit integer	Height of bi-level layer band, in black pixels.
bi-level layer band data size	32-bit integer	Size of bi-level layer band data, in bytes.
contone band height	16-bit integer	Height of contone band, in contone pixels.
contone band data size	32-bit integer	Size of contone plane band data, in bytes.
tag band height	16-bit integer	Height of tag band, in dots.
tag band data size	32-bit integer	Size of unencoded tag data band, in bytes. Can be 0 which indicates that no tag data is provided.
reserved	up to 128 bytes	Reserved and 0 pads out band header to multiple of 128 bytes.

The bi-level layer parameters define the height of the black band, and the size of its compressed band data. The variable-size black data follows the page band header.

The contone layer parameters define the height of the contone band, and the size of its compressed page data. The variable-size contone data follows the black data.

The tag band data is the set of variable tag data half-lines as required by the tag encoder. The format of the tag data is found in Section 26.5.2. The tag band data follows the contone data.

Table 5 shows the format of the variable-size compressed band data which follows the page band header.

Table 5. Page band data format

field	format	Description
black data	Modified G4 facsimile bitstream[4]	Compressed bi-level layer.
contone data	JPEG bytestream	Compressed contone datalayer.
tag data map	Tag data array	Tag data format. See Section 26.5.2.

The start of each variable-size segment of band data should be aligned to a 256-bit DRAM word boundary.

The following sections describe the format of the compressed bi-level layers and the compressed contone layer. section 26.5.1 on page 410 describes the format of the tag data structures.

8.1.2.3   Bi-level data compression

The (typically 1600 dpi) black bi-level layer is losslessly compressed using Silverbrook Modified Group 4 (SMG4) compression which is a version of Group 4 Facsimile compression [22] without Huffman and with simplified run length encodings. Typically compression ratios exceed 10:1. The encoding are listed in Table 6 and Table 7.

Table 6. Bi-Level group 4 facsimile style compression encodings

	Encoding	Description
same as Group 4	1000	Pass Command: a0 ← b2, skip next two edges

---
[4]   See section 8.1.2.3 on page 36 for note regarding the use of this standard

Facsimile		
	1	Vertical(0): a0 ← b1, color = !color
	110	Vertical(1): a0 ← b1 + 1, color = !color
	010	Vertical(-1): a0 ← b1 - 1, color = !color
	110000	Vertical(2): a0 ← b1 + 2, color = !color
	010000	Vertical(-2): a0 ← b1 - 2, color = !color
Unique to this implementation	100000	Vertical(3): a0 ← b1 + 3, color = !color
	000000	Vertical(-3): a0 ← b1 - 3, color = !color
	<RL><RL>100	Horizontal: a0 ← a0 + <RL> + <RL>

SMG4 has a pass through mode to cope with local negative compression. Pass through mode is activated by a special run-length code. Pass through mode continues to either end of line or for a pre-programmed number of bits, whichever is shorter. The special run-length code is always executed as a run-length code, followed by pass through. The pass through escape code is a
5   medium length run-length with a run of less than or equal to 31.

Table 7. Run length (RL) encodings

	Encoding	Description
Unique to this implementation	RRRRR1	Short Black Runlength (5 bits)
	RRRRR1	Short White Runlength (5 bits)
	RRRRRRRRRR10	Medium Black Runlength (10 bits)
	RRRRRRRR10	Medium White Runlength (8 bits)
	RRRRRRRRRR10	Medium Black Runlength with RRRRRRRRRR <= 31, Enter pass through
	RRRRRRRR10	Medium White Runlength with RRRRRRRR <= 31, Enter pass through
	RRRRRRRRRRRRRRR00	Long Black Runlength (15 bits)
	RRRRRRRRRRRRRRR00	Long White Runlength (15 bits)

Since the compression is a bitstream, the encodings are read right (least significant bit) to left (most significant bit). The run lengths given as RRRR in Table   are read in the same way (least
10   significant bit at the right to most significant bit at the left).

Each band of bi-level data is optionally self contained. The first line of each band therefore is based on a 'previous' blank line or the last line of the previous band.

8.1.2.3.1 Group 3 and 4 facsimile compression

The Group 3 Facsimile compression algorithm [22] losslessly compresses bi-level data for
15   transmission over slow and noisy telephone lines. The bi-level data represents scanned black text and graphics on a white background, and the algorithm is tuned for this class of images (it is explicitly not tuned, for example, for *halftoned* bi-level images). The *1D* Group 3 algorithm

runlength-encodes each scanline and then Huffman-encodes the resulting runlengths. Runlengths in the range 0 to 63 are coded with *terminating* codes. Runlengths in the range 64 to 2623 are coded with *make-up* codes, each representing a multiple of 64, followed by a terminating code. Runlengths exceeding 2623 are coded with multiple make-up codes followed by a terminating code.

5   The Huffman tables are fixed, but are separately tuned for black and white runs (except for make-up codes above 1728, which are common). When possible, the *2D* Group 3 algorithm encodes a scanline as a set of short edge deltas (0, ±1, ±2, ±3) with reference to the previous scanline. The delta symbols are entropy-encoded (so that the zero delta symbol is only one bit long etc.) Edges within a 2D-encoded line which can't be delta-encoded are runlength-encoded, and are identified by

10   a prefix. 1D- and 2D-encoded lines are marked differently. 1D-encoded lines are generated at regular intervals, whether actually required or not, to ensure that the decoder can recover from line noise with minimal image degradation. 2D Group 3 achieves compression ratios of up to 6:1 [32]. The Group 4 Facsimile algorithm [22] losslessly compresses bi-level data for transmission over *error-free* communications lines (i.e. the lines are truly error-free, or error-correction is done at a

15   lower protocol level). The Group 4 algorithm is based on the 2D Group 3 algorithm, with the essential modification that since transmission is assumed to be error-free, 1D-encoded lines are no longer generated at regular intervals as an aid to error-recovery. Group 4 achieves compression ratios ranging from 20:1 to 60:1 for the CCITT set of test images [32].

The design goals and performance of the Group 4 compression algorithm qualify it as a

20   compression algorithm for the bi-level layers. However, its Huffman tables are tuned to a lower scanning resolution (100-400 dpi), and it encodes runlengths exceeding 2623 awkwardly.

8.1.2.4   Contone data compression

The contone layer (CMYK) is either a non-compressed bytestream or is compressed to an interleaved JPEG bytestream. The JPEG bytestream is complete and self-contained. It contains all

25   data required for decompression, including quantization and Huffman tables.

The contone data is optionally converted to YCrCb before being compressed (there is no specific advantage in color-space converting if not compressing). Additionally, the CMY contone pixels are optionally converted (on an individual basis) to RGB before color conversion using R=255-C, G=255-M, B=255-Y. Optional bitwise inversion of the K plane may also be performed. Note that this

30   CMY to RGB conversion is not intended to be accurate for display purposes, but rather for the purposes of later converting to YCrCb. The inverse transform will be applied before printing.

8.1.2.4.1 JPEG compression

The JPEG compression algorithm [27] lossily compresses a contone image at a specified quality level. It introduces imperceptible image degradation at compression ratios below 5:1, and negligible

35   image degradation at compression ratios below 10:1 [33].

JPEG typically first transforms the image into a color space which separates luminance and chrominance into separate color channels. This allows the chrominance channels to be subsampled without appreciable loss because of the human visual system's relatively greater sensitivity to luminance than chrominance. After this first step, each color channel is compressed separately.

The image is divided into 8×8 pixel blocks. Each block is then transformed into the frequency domain via a discrete cosine transform (DCT). This transformation has the effect of concentrating image energy in relatively lower-frequency coefficients, which allows higher-frequency coefficients to be more crudely quantized. This quantization is the principal source of compression in JPEG.

5 Further compression is achieved by ordering coefficients by frequency to maximize the likelihood of adjacent zero coefficients, and then runlength-encoding runs of zeroes. Finally, the runlengths and non-zero frequency coefficients are entropy coded. Decompression is the inverse process of compression.

8.1.2.4.2 Non-compressed format

10 If the contone data is non-compressed, it must be in a block-based format bytestream with the same pixel order as would be produced by a JPEG decoder. The bytestream therefore consists of a series of 8×8 block of the original image, starting with the top left 8×8 block, and working horizontally across the page (as it will be printed) until the top rightmost 8×8 block, then the next row of 8×8 blocks (left to right) and so on until the lower row of 8×8 blocks (left to right). Each 8×8

15 block consists of 64 8-bit pixels for color plane 0 (representing 8 rows of 8 pixels in the order top left to bottom right) followed by 64 8-bit pixels for color plane 1 and so on for up to a maximum of 4 color planes.

If the original image is not a multiple of 8 pixels in X or Y, padding must be present (the extra pixel data will be ignored by the setting of margins).

20 8.1.2.4.3 Compressed format

If the contone data is compressed the first memory band contains JPEG headers (including tables) plus MCUs (minimum coded units). The ratio of space between the various color planes in the JPEG stream is 1:1:1:1. No subsampling is permitted. Banding can be completely arbitrary i.e there can be multiple JPEG images per band or 1 JPEG image divided over multiple bands. The break

25 between bands is only memory alignment based.

8.1.2.4.4 Conversion of RGB to YCrCb (in RIP)

YCrCb is defined as per CCIR 601-1 [24] except that Y, Cr and Cb are normalized to occupy all 256 levels of an 8-bit binary encoding and take account of the actual hardware implementation of the inverse transform within SoPEC.

30 The exact color conversion computation is as follows:

- $Y^* = (9805/32768)R + (19235/32768)G + (3728/32768)B$
- $Cr^* = (16375/32768)R - (13716/32768)G - (2659/32768)B + 128$
- $Cb^* = -(5529/32768)R - (10846/32768)G + (16375/32768)B + 128$

Y, Cr and Cb are obtained by rounding to the nearest integer. There is no need for saturation since

35 ranges of Y^*, Cr^* and Cb^* after rounding are [0-255], [1-255] and [1-255] respectively. *Note that full accuracy is possible with 24 bits.* See [14] for more information.

SoPEC ASIC

9　Overview

The Small Office Home Office Print Engine Controller (SoPEC) is a page rendering engine ASIC

40 that takes compressed page images as input, and produces decompressed page images at up to 6

channels of bi-level dot data as output. The bi-level dot data is generated for the Memjet bi-lithic printhead. The dot generation process takes account of printhead construction, dead nozzles, and allows for fixative generation.

A single SoPEC can control 2 bi-lithic printheads and up to 6 color channels at 10,000 lines/sec[5], equating to 30 pages per minute. A single SoPEC can perform full-bleed printing of A3, A4 and Letter pages. The 6 channels of colored ink are the expected maximum in a consumer SOHO, or office Bi-lithic printing environment:

- CMY, for regular color printing.
- K, for black text, line graphics and gray-scale printing.
- IR (infrared), for Netpage-enabled [5] applications.
- F (fixative), to enable printing at high speed. Because the bi-lithic printer is capable of printing so fast, a fixative may be required to enable the ink to dry before the page touches the page already printed. Otherwise the pages may bleed on each other. In low speed printing environments the fixative may not be required.

SoPEC is *color space agnostic*. Although it can accept contone data as CMYX or RGBX, where X is an optional 4th channel, it also can accept contone data in any print color space. Additionally, SoPEC provides a mechanism for arbitrary mapping of input channels to output channels, including combining dots for ink optimization, generation of channels based on any number of other channels etc. However, inputs are typically CMYK for contone input, K for the bi-level input, and the optional Netpage tag dots are typically rendered to an infra-red layer. A fixative channel is typically generated for fast printing applications.

SoPEC is *resolution agnostic*. It merely provides a mapping between input resolutions and output resolutions by means of scale factors. The expected output resolution is 1600 dpi, but SoPEC actually has no knowledge of the physical resolution of the Bi-lithic printhead.

SoPEC is *page-length agnostic*. Successive pages are typically split into bands and downloaded into the page store as each band of information is consumed and becomes free.

SoPEC provides an interface for synchronization with other SoPECs. This allows simple multi-SoPEC solutions for simultaneous A3/A4/Letter duplex printing. However, SoPEC is also capable of printing only a portion of a page image. Combining synchronization functionality with partial page rendering allows multiple SoPECs to be readily combined for alternative printing requirements including simultaneous duplex printing and wide format printing.

Table 8 lists some of the features and corresponding benefits of SoPEC.

Table 8. Features and Benefits of SoPEC

Feature	Benefits
Optimised print architecture in hardware	30ppm full page photographic quality color printing from a desktop PC

---

[5]10,000 lines per second equates to 30 A4/Letter pages per minute at 1600 dpi

0.13micron CMOS (>3 million transistors)	High speed Low cost High functionality
900 Million dots per second	Extremely fast page generation
10,000 lines per second at 1600 dpi	0.5 A4/Letter pages per SoPEC chip per second
1 chip drives up to 133,920 nozzles	Low cost page-width printers
1 chip drives up to 6 color planes	99% of SoHo printers can use 1 SoPEC device
Integrated DRAM	No external memory required, leading to low cost systems
Power saving sleep mode	SoPEC can enter a power saving sleep mode to reduce power dissipation between print jobs
JPEG expansion	Low bandwidth from PC Low memory requirements in printer
Lossless bitplane expansion	High resolution text and line art with low bandwidth from PC (e.g. over USB)
Netpage tag expansion	Generates interactive paper
Stochastic dispersed dot dither	Optically smooth image quality No moire effects
Hardware compositor for 6 image planes	Pages composited in real-time
Dead nozzle compensation	Extends printhead life and yield Reduces printhead cost
Color space agnostic	Compatible with all inksets and image sources including RGB, CMYK, spot, CIE L*a*b*, hexachrome, YCrCbK, sRGB and other
Color space conversion	Higher quality / lower bandwidth
Computer interface	USB1.1 interface to host and ISI interface to ISI-Bridge chip thereby allowing connection to IEEE 1394, Bluetooth etc.
Cascadable in resolution	Printers of any resolution
Cascadable in color depth	Special color sets e.g. hexachrome can be used
Cascadable in image size	Printers of any width up to 16 inches
Cascadable in pages	Printers can print both sides simultaneously
Cascadable in speed	Higher speeds are possible by having each SoPEC print one vertical strip of the page.
Fixative channel data generation	Extremely fast ink drying without wastage
Built-in security	Revenue models are protected
Undercolor removal on dot-by-dot basis	Reduced ink usage

Does not require fonts for high speed operation	No font substitution or missing fonts
Flexible printhead configuration	Many configurations of printheads are supported by one chip type
Drives Bi-lithic printheads directly	No print driver chips required, results in lower cost
Determines dot accurate ink usage	Removes need for physical ink monitoring system in ink cartridges

9.1 PRINTING RATES

The required printing rate for SoPEC is 30 sheets per minute with an inter-sheet spacing of 4 cm. To achieve a 30 sheets per minute print rate, this requires:

300mm x 63 (dot/mm) / 2 sec = 105.8 μseconds per line, with no inter-sheet gap.

340mm x 63 (dot/mm) / 2 sec = 93.3 μseconds per line, with a 4 cm inter-sheet gap.

A printline for an A4 page consists of 13824 nozzles across the page [2]. At a system clock rate of 160 MHz 13824 dots of data can be generated in 86.4 μseconds. Therefore data can be generated fast enough to meet the printing speed requirement. It is necessary to deliver this print data to the print-heads.

Printheads can be made up of 5:5, 6:4, 7:3 and 8:2 inch printhead combinations [2]. Print data is transferred to both print heads in a pair simultaneously. This means the longest time to print a line is determined by the time to transfer print data to the longest print segment. There are 9744 nozzles across a 7 inch printhead. The print data is transferred to the printhead at a rate of 106 MHz (2/3 of the system clock rate) per color plane. This means that it will take 91.9 μs to transfer a single line for a 7:3 printhead configuration. So we can meet the requirement of 30 sheets per minute printing with a 4 cm gap with a 7:3 printhead combination. There are 11160 across an 8 inch printhead. To transfer the data to the printhead at 106 MHz will take 105.3 μs. So an 8:2 printhead combination printing with an inter-sheet gap will print slower than 30 sheets per minute.

9.2 SoPEC BASIC ARCHITECTURE

From the highest point of view the SoPEC device consists of 3 distinct subsystems

- CPU Subsystem
- DRAM Subsystem
- Print Engine Pipeline (PEP) Subsystem

See Figure 13 for a block level diagram of SoPEC.

9.2.1 CPU Subsystem

The CPU subsystem controls and configures all aspects of the other subsystems. It provides general support for interfacing and synchronising the external printer with the internal print engine. It also controls the low speed communication to the QA chips. The CPU subsystem contains various peripherals to aid the CPU, such as GPIO (includes motor control), interrupt controller, LSS Master and general timers. The Serial Communications Block (SCB) on the CPU subsystem provides a full speed USB1.1 interface to the host as well as an Inter SoPEC Interface (ISI) to other SoPEC devices.

9.2.2 DRAM Subsystem

The DRAM subsystem accepts requests from the CPU, Serial Communications Block (SCB) and blocks within the PEP subsystem. The DRAM subsystem (in particular the DIU) arbitrates the various requests and determines which request should win access to the DRAM. The DIU arbitrates based on configured parameters, to allow sufficient access to DRAM for all requestors. The DIU
5   also hides the implementation specifics of the DRAM such as page size, number of banks, refresh rates etc.

9.2.3   Print Engine Pipeline (PEP) subsystem

The Print Engine Pipeline (PEP) subsystem accepts compressed pages from DRAM and renders them to bi-level dots for a given print line destined for a printhead interface that communicates
10   directly with up to 2 segments of a bi-lithic printhead.

The first stage of the page expansion pipeline is the CDU, LBD and TE. The CDU expands the JPEG-compressed contone (typically CMYK) layer, the LBD expands the compressed bi-level layer (typically K), and the TE encodes Netpage tags for later rendering (typically in IR or K ink). The output from the first stage is a set of buffers: the CFU, SFU, and TFU. The CFU and SFU buffers
15   are implemented in DRAM.

The second stage is the HCU, which dithers the contone layer, and composites position tags and the bi-level spot0 layer over the resulting bi-level dithered layer. A number of options exist for the way in which compositing occurs. Up to 6 channels of bi-level data are produced from this stage. Note that not all 6 channels may be present on the printhead. For example, the printhead may be
20   CMY only, with K pushed into the CMY channels and IR ignored. Alternatively, the position tags may be printed in K if IR ink is not available (or for testing purposes).

The third stage (DNC) compensates for dead nozzles in the printhead by color redundancy and error diffusing dead nozzle data into surrounding dots.

The resultant bi-level 6 channel dot-data (typically CMYK-IRF) is buffered and written out to a set of
25   line buffers stored in.DRAM via the DWU.

Finally, the dot-data is loaded back from DRAM, and passed to the printhead interface via a dot FIFO. The dot FIFO accepts data from the LLU at the system clock rate ($pclk$), while the PHI removes data from the FIFO and sends it to the printhead at a rate of 2/3 times the system clock rate (see Section 9.1).

30

9.3   SoPEC Block Description

Looking at Figure 13, the various units are described here in summary form:

Table 9. Units within SoPEC

Subsystem	Unit Acronym	Unit Name	Description
DRAM	DIU	DRAM interface unit	Provides the interface for DRAM read and write access for the various SoPEC units, CPU and the SCB block. The DIU provides arbitration between competing units controls DRAM

			access.
	DRAM	Embedded DRAM	20Mbits of embedded DRAM,
CPU	CPU	Central Processing Unit	CPU for system configuration and control
	MMU	Memory Management Unit	Limits access to certain memory address areas in CPU user mode
	RDU	Real-time Debug Unit	Facilitates the observation of the contents of most of the CPU addressable registers in SoPEC in addition to some pseudo-registers in realtime.
	TIM	General Timer	Contains watchdog and general system timers
	LSS	Low Speed Serial Interfaces	Low level controller for interfacing with the QA chips
	GPIO	General Purpose IOs	General IO controller, with built-in Motor control unit, LED pulse units and de-glitch circuitry
	ROM	Boot ROM	16 KBytes of System Boot ROM code
	ICU	Interrupt Controller Unit	General Purpose interrupt controller with configurable priority, and masking.
	CPR	Clock, Power and Reset block	Central Unit for controlling and generating the system clocks and resets and powerdown mechanisms
	PSS	Power Save Storage	Storage retained while system is powered down
	USB	Universal Serial Bus Device	USB device controller for interfacing with the host USB.
	ISI	Inter-SoPEC Interface	ISI controller for data and control communication with other SoPEC's in a multi-SoPEC system
	SCB	Serial Communication Block	Contains both the USB and ISI blocks.
Print Engine Pipeline (PEP)	PCU	PEP controller	Provides external CPU with the means to read and write PEP Unit registers, and read and write DRAM in single 32-bit chunks.
	CDU	Contone decoder unit	Expands JPEG compressed contone layer and writes decompressed contone to DRAM
	CFU	Contone FIFO Unit	Provides line buffering between CDU and HCU
	LBD	Lossless Bi-level Decoder	Expands compressed bi-level layer.
	SFU	Spot FIFO Unit	Provides line buffering between LBD and HCU
	TE	Tag encoder	Encodes tag data into line of tag dots.

TFU	Tag FIFO Unit	Provides tag data storage between TE and HCU
HCU	Halftoner compositor unit	Dithers contone layer and composites the bi-level spot 0 and position tag dots.
DNC	Dead Nozzle Compensator	Compensates for dead nozzles by color redundancy and error diffusing dead nozzle data into surrounding dots.
DWU	Dotline Writer Unit	Writes out the 6 channels of dot data for a given printline to the line store DRAM
LLU	Line Loader Unit	Reads the expanded page image from line store, formatting the data appropriately for the bi-lithic printhead.
PHI	PrintHead Interface	Is responsible for sending dot data to the bi-lithic printheads and for providing line synchronization between multiple SoPECs. Also provides test interface to printhead such as temperature monitoring and Dead Nozzle Identification.

9.4   ADDRESSING SCHEME IN SoPEC

SoPEC must address

•   20 Mbit DRAM.

5   •   PCU addressed registers in PEP.

•   CPU-subsystem addressed registers.

SoPEC has a unified address space with the CPU capable of addressing all CPU-subsystem and PCU-bus accessible registers (in PEP) and all locations in DRAM. The CPU generates byte-aligned addresses for the whole of SoPEC.

10   22 bits are sufficient to byte address the whole SoPEC address space.

9.4.1   DRAM addressing scheme

The embedded DRAM is composed of 256-bit words. However the CPU-subsystem may need to write individual bytes of DRAM. Therefore it was decided to make the DIU byte addressable. 22 bits are required to byte address 20 Mbits of DRAM.

15   Most blocks read or write 256-bit words of DRAM. Therefore only the top 17 bits i.e. bits 21 to 5 are required to address 256-bit word aligned locations.

The exceptions are

•   CDU which can write 64-bits so only the top 19 address bits i.e. bits 21-3 are required.

•   The CPU-subsystem always generates a 22-bit byte-aligned DIU address but it will send

20   flags to the DIU indicating whether it is an 8, 16 or 32-bit write.

All DIU accesses must be within the same 256-bit aligned DRAM word.

9.4.2   PEP Unit DRAM addressing

PEP Unit configuration registers which specify DRAM locations should specify 256-bit aligned DRAM addresses i.e. using address bits 21:5. Legacy blocks from PEC1 e.g. the LBD and TE may need to specify 64-bit aligned DRAM addresses if these reused blocks DRAM addressing is difficult to modify. These 64-bit aligned addresses require address bits 21:3. However, these 64-bit aligned

5   addresses should be programmed to start at a 256-bit DRAM word boundary.

Unlike PEC1, there are no constraints in SoPEC on data organization in DRAM except that all data structures must start on a 256-bit DRAM boundary. If data stored is not a multiple of 256-bits then the last word should be padded.

9.4.3   CPU subsystem bus addressed registers

10   The CPU subsystem bus supports 32-bit word aligned read and write accesses with variable access timings. See section 11.4 for more details of the access protocol used on this bus. The CPU subsystem bus does not currently support byte reads and writes but this can be added at a later date if required by imported IP.

9.4.4   PCU addressed registers in PEP

15   The PCU only supports 32-bit register reads and writes for the PEP blocks. As the PEP blocks only occupy a subsection of the overall address map and the PCU is explicitly selected by the MMU when a PEP block is being accessed the PCU does not need to perform a decode of the higher-order address bits. See Table 11 for the PEP subsystem address map.

9.5   SoPEC MEMORY MAP

20   ### 9.5.1   Main memory map

The system wide memory map is shown in Figure 14 below. The memory map is discussed in detail in Section 11 11 Central Processing Unit (CPU).

9.5.2   CPU-bus peripherals address map

The address mapping for the peripherals attached to the CPU-bus is shown in Table 10 below. The

25   MMU performs the decode of cpu_adr[21:12] to generate the relevant cpu_block_select signal for each block. The addressed blocks decode however many of the lower order bits of cpu_adr[11:2] are required to address all the registers within the block.

Table 10. CPU-bus peripherals address map

Block_base	Address
ROM_base	0x0000_0000
MMU_base	0x0001_0000
TIM_base	0x0001_1000
LSS_base	0x0001_2000
GPIO_base	0x0001_3000
SCB_base	0x0001_4000
ICU_base	0x0001_5000
CPR_base	0x0001_6000
DIU_base	0x0001_7000

PSS_base	0x0001_8000
Reserved	0x0001_9000 to 0x0001_FFFF
PCU_base	0x0002_0000 to 0x0002_BFFF

9.5.3   PCU Mapped Registers (PEP blocks) address map

The PEP blocks are addressed via the PCU. From Figure 14, the PCU mapped registers are in the range 0x0002_0000 to 0x0002_BFFF. From Table 11 it can be seen that there are 12 sub-blocks within the PCU address space. Therefore, only four bits are necessary to address each of the sub-

5   blocks within the PEP part of SoPEC. A further 12 bits may be used to address any configurable register within a PEP block. This gives scope for 1024 configurable registers per sub-block (the PCU mapped registers are all 32-bit addressed registers so the upper 10 bits are required to individually address them). This address will come either from the CPU or from a command stored in DRAM. The bus is assembled as follows:

10 •   address[15:12] = sub-block address,

•   address[n:2] = register address within sub-block, only the number of bits required to decode the registers within each sub-block are used,

•   address[1:0] = byte address, unused as PCU mapped registers are all 32-bit addressed registers.

15   So for the case of the HCU, its addresses range from 0x7000 to 0x7FFF within the PEP subsystem or from 0x0002_7000 to 0x0002_7FFF in the overall system.

Table 11. PEP blocks address map

Block_base	Address
PCU_base	0x0002_0000
CDU_base	0x0002_1000
CFU_base	0x0002_2000
LBD_base	0x0002_3000
SFU_base	0x0002_4000
TE_base	0x0002_5000
TFU_base	0x0002_6000
HCU_base	0x0002_7000
DNC_base	0x0002_8000
DWU_base	0x0002_9000
LLU_base	0x0002_A000
PHI_base	0x0002_B000 to 0x0002_BFFF

9.6   BUFFER MANAGEMENT IN SoPEC

20   As outlined in Section 9.1, SoPEC has a requirement to print 1 side every 2 seconds i.e. 30 sides per minute.

9.6.1   Page buffering

Approximately 2 Mbytes of DRAM are reserved for compressed page buffering in SoPEC. If a page is compressed to fit within 2 Mbyte then a complete page can be transferred to DRAM before printing. However, the time to transfer 2 Mbyte using USB 1.1 is approximately 2 seconds. The worst case cycle time to print a page then approaches 4 seconds. This reduces the worst-case print speed to 15 pages per minute.

9.6.2 Band buffering

The SoPEC page-expansion blocks support the notion of page banding. The page can be divided into bands and another band can be sent down to SoPEC while we are printing the current band. Therefore we can start printing once at least one band has been downloaded.

The band size granularity should be carefully chosen to allow efficient use of the USB bandwidth and DRAM buffer space. It should be small enough to allow seamless 30 sides per minute printing but not so small as to introduce excessive CPU overhead in orchestrating the data transfer and parsing the band headers. Band-finish interrupts have been provided to notify the CPU of free buffer space. It is likely that the host PC will supervise the band transfer and buffer management instead of the SoPEC CPU.

If SoPEC starts printing before the complete page has been transferred to memory there is a risk of a buffer underrun occurring if subsequent bands are not transferred to SoPEC in time e.g. due to insufficient USB bandwidth caused by another USB peripheral consuming USB bandwidth. A buffer underrun occurs if a line synchronisation pulse is received before a line of data has been transferred to the printhead and causes the print job to fail at that line. If there is no risk of buffer underrun then printing can safely start once at least one band has been downloaded.

If there is a risk of a buffer underrun occurring due to an interruption of compressed page data transfer, then the safest approach is to only start printing once we have loaded up the data for a complete page. This means that a worst case latency in the region of 2 seconds (with USB1.1) will be incurred before printing the first page. Subsequent pages will take 2 seconds to print giving us the required sustained printing rate of 30 sides per minute.

A Storage SoPEC (Section 7.2.5) could be added to the system to provide guaranteed bandwidth data delivery. The print system could also be constructed using an ISI-Bridge chip (Section 7.2.6) to provide guaranteed data delivery.

The most efficient page banding strategy is likely to be determined on a per page/ print job basis and so SoPEC will support the use of bands of any size.

10 SoPEC Use Cases

10.1 INTRODUCTION

This chapter is intended to give an overview of a representative set of scenarios or *use cases* which SoPEC can perform. SoPEC is by no means restricted to the particular use cases described and not every SoPEC system is considered here.

In this chapter we discuss SoPEC use cases under four headings:

1) Normal operation use cases.

2) Security use cases.

3) Miscellaneous use cases.

4) Failure mode use cases.

Use cases for both single and multi-SoPEC systems are outlined.

Some tasks may be composed of a number of sub-tasks.

The realtime requirements for SoPEC software tasks are discussed in " 11 Central Processing Unit (CPU)" under Section 11.3 Realtime requirements.

10.2 NORMAL OPERATION IN A SINGLE SOPEC SYSTEM WITH USB HOST CONNECTION

SoPEC operation is broken up into a number of sections which are outlined below. Buffer management in a SoPEC system is normally performed by the host.

10.2.1   Powerup

Powerup describes SoPEC initialisation following an external reset or the watchdog timer system reset.

A typical powerup sequence is:

1)   Execute reset sequence for complete SoPEC.

2)   CPU boot from ROM.

3)   Basic configuration of CPU peripherals, SCB and DIU. DRAM initialisation. USB Wakeup.

4)   Download and authentication of program (see Section 10.5.2).

5)   Execution of program from DRAM.

6)   Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.

7)   Download and authenticate any further *datasets*.

10.2.2   USB wakeup

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block (chapter 16). Normally the CPU sub-system and the DRAM will be put in sleep mode but the SCB and power-safe storage (PSS) will still be enabled.

Wakeup describes SoPEC recovery from sleep mode with the SCB and power-safe storage (PSS) still enabled. In a single SoPEC system, wakeup can be initiated following a USB reset from the SCB.

A typical USB wakeup sequence is:

1)   Execute reset sequence for sections of SoPEC in sleep mode.

2)   CPU boot from ROM, if CPU-subsystem was in sleep mode.

3)   Basic configuration of CPU peripherals and DIU, and DRAM initialisation, if required.

4)   Download and authentication of program using results in Power-Safe Storage (PSS) (see Section 10.5.2).

5)   Execution of program from DRAM.

6)   Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.

7)   Download and authenticate using results in PSS of any further *datasets* (programs).

10.2.3   Print initialization

This sequence is typically performed at the start of a print job following powerup or wakeup:

1)   Check amount of ink remaining via QA chips.

2)   Download static data e.g. dither matrices, dead nozzle tables from host to DRAM.

3) Check printhead temperature, if required, and configure printhead with firing pulse profile etc. accordingly.

4) Initiate printhead pre-heat sequence, if required.

10.2.4   First page download

Buffer management in a SoPEC system is normally performed by the host.

First page, first band download and processing:

1) The host communicates to the SoPEC CPU over the USB to check that DRAM space remaining is sufficient to download the first band.

2) The host downloads the first band (with the page header) to DRAM.

3) When the complete page header has been downloaded the SoPEC CPU processes the page header, calculates PEP register commands and writes directly to PEP registers or to DRAM.

4)   If PEP register commands have been written to DRAM, execute PEP commands from DRAM via PCU.

Remaining bands download and processing:

1) Check DRAM space remaining is sufficient to download the next band.

2) Download the next band with the band header to DRAM.

3) When the complete band header has been downloaded, process the band header according to whichever band-related register updating mechanism is being used.

10.2.5   Start printing

1) Wait until at least one band of the first page has been downloaded.   .

One approach is to only start printing once we have loaded up the data for a complete page. If we start printing before the complete page has been transferred to memory we run the risk of a buffer underrun occurring because compressed page data was not transferred to SoPEC in time e.g. due to insufficient USB bandwidth caused by another USB peripheral consuming USB bandwidth.

2) Start all the PEP Units by writing to their *Go* registers, via PCU commands executed from DRAM or direct CPU writes. A rapid startup order for the PEP units is outlined in Table 12.

Table 12. Typical PEP Unit startup order for printing a page.

Step#	Unit
1	DNC
2	DWU
3	HCU
4	PHI
5	LLU
6	CFU, SFU, TFU
7	CDU
8	TE, LBD

3) Print ready interrupt occurs (from PHI).

4) Start motor control, if first page, otherwise feed the next page. This step could occur before the print ready interrupt.

5) Drive LEDs, monitor paper status.

6) Wait for page alignment via page sensor(s) GPIO interrupt.

7) CPU instructs PHI to start producing line syncs and hence commence printing, or wait for an external device to produce line syncs.

8) Continue to download bands and process page and band headers for next page.

10.2.6   Next page(s) download

As for first page download, performed during printing of current page.

10.2.7   Between bands

When the finished band flags are asserted band related registers in the CDU, LBD, TE need to be re-programmed before the subsequent band can be printed. This can be via PCU commands from DRAM. Typically only 3-5 commands per decompression unit need to be executed. These registers can also be reprogrammed directly by the CPU or most likely by updating from shadow registers. The finished band flag interrupts the CPU to tell the CPU that the area of memory associated with the band is now free.

10.2.8   During page print

Typically during page printing ink usage is communicated to the QA chips.

1) Calculate ink printed (from PHI).

2) Decrement ink remaining (via QA chips).

3) Check amount of ink remaining (via QA chips). This operation may be better performed while the page is being printed rather than at the end of the page.

10.2.9   Page finish

These operations are typically performed when the page is finished:

1) Page finished interrupt occurs from PHI.

2) Shutdown the PEP blocks by de-asserting their Go registers. A typical shutdown order is defined in Table 13. This will set the PEP Unit state-machines to their idle states without resetting their configuration registers.

3) Communicate ink usage to QA chips, if required.

Table 13. End of page shutdown order for PEP Units.

Step#	Unit
1	PHI (will shutdown by itself in the normal case at the end of a page)
2	DWU (shutting this down stalls the DNC and therefore the HCU and above)
3	LLU (should already be halted due to PHI at end of last line of page)
4	TE (this is the only dot supplier likely to be running, halted by the HCU)
5	CDU (this is likely to already be halted due to end of contone band)

| 6 | CFU, SFU, TFU, LBD (order unimportant, and should already be halted due to end of band) |
| 7 | HCU, DNC (order unimportant, should already have halted) |

10.2.10 Start of next page

These operations are typically performed before printing the next page:

1) Re-program the PEP Units via PCU command processing from DRAM based on page header.

2) Go to Start printing.

10.2.11 End of document

1) Stop motor control.

10.2.12 Sleep mode

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block described in Section 16.

1) Instruct host PC via USB that SoPEC is about to sleep.

2) Store reusable authentication results in Power-Safe Storage (PSS).

3) Put SoPEC into defined sleep mode.

10.3 NORMAL OPERATION IN A MULTI-SOPEC SYSTEM - ISIMASTER SOPEC

In a multi-SoPEC system the host generally manages program and compressed page download to all the SoPECs. Inter-SoPEC communication is over the ISI link which will add a latency.

In the case of a multi-SoPEC system with just one USB 1.1 connection, the SoPEC with the USB connection is the ISIMaster. The ISI-bridge chip is the ISIMaster in the case of an ISI-Bridge SoPEC configuration. While it is perfectly possible for an ISISlave to have a direct USB connection to the host we do not treat this scenario explicitly here to avoid possible confusion.

In a multi-SoPEC system one of the SoPECs will be the PrintMaster. This SoPEC must manage and control sensors and actuators e.g. motor control. These sensors and actuators could be distributed over all the SoPECs in the system. An ISIMaster SoPEC may also be the PrintMaster SoPEC.

In a multi-SoPEC system each printing SoPEC will generally have its own PRINTER_QA chip (or at least access to a PRINTER_QA chip that contains the SoPEC's SOPEC_id_key) to validate operating parameters and ink usage. The results of these operations may be communicated to the PrintMaster SoPEC.

In general the ISIMaster may need to be able to:

• Send messages to the ISISlaves which will cause the ISISlaves to send their status to the ISIMaster.

• Instruct the ISISlaves to perform certain operations.

As the ISI is an insecure interface commands issued over the ISI are regarded as *user mode* commands. *Supervisor mode* code running on the SoPEC CPUs will allow or disallow these commands. The software protocol needs to be constructed with this in mind.

The ISIMaster will initiate all communication with the ISISlaves.

SoPEC operation is broken up into a number of sections which are outlined below.

10.3.1 Powerup

Powerup describes SoPEC initialisation following an external reset or the watchdog timer system reset.

1) Execute reset sequence for complete SoPEC.

2) CPU boot from ROM.

3) Basic configuration of CPU peripherals, SCB and DIU. DRAM initialisation USB Wakeup

4) SoPEC identification by activity on USB end-points 2-4 indicates it is the ISIMaster (unless the SoPEC CPU has explicitly disabled this function).

5) Download and authentication of program (see Section 10.5.3).

6) Execution of program from DRAM.

7) Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.

8) Download and authenticate any further *datasets* (programs).

9) The initial *dataset* may be broadcast to all the ISISlaves.

10) ISIMaster master SoPEC then waits for a short time to allow the authentication to take place on the ISISlave SoPECs.

11) Each ISISlave SoPEC is polled for the result of its program code authentication process.

12) If all ISISlaves report successful authentication the OEM code module can be distributed and authenticated. OEM code will most likely reside on one SoPEC.

10.3.2   USB wakeup

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block [16]. Normally the CPU sub-system and the DRAM will be put in sleep mode but the SCB and power-safe storage (PSS) will still be enabled.

Wakeup describes SoPEC recovery from sleep mode with the SCB and power-safe storage (PSS) still enabled. For an ISIMaster SoPEC connected to the host via USB, wakeup can be initiated following a USB reset from the SCB.

A typical USB wakeup sequence is:

1) Execute reset sequence for sections of SoPEC in sleep mode.

2) CPU boot from ROM, if CPU-subsystem was in sleep mode.

3) Basic configuration of CPU peripherals and DIU, and DRAM initialisation, if required.

4) SoPEC identification by activity on USB end-points 2-4 indicates it is the ISIMaster (unless the SoPEC CPU has explicitly disabled this function).

5) Download and authentication of program using results in Power-Safe Storage (PSS) (see Section 10.5.3).

6) Execution of program from DRAM.

7) Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.

8) Download and authenticate any further *datasets* (programs) using results in Power-Safe Storage (PSS) (see Section 10.5.3).

9) Following steps as per Powerup.

10.3.3   Print initialization

This sequence is typically performed at the start of a print job following powerup or wakeup:

1) Check amount of ink remaining via QA chips which may be present on a ISISlave SoPEC.

2) Download static data e.g. dither matrices, dead nozzle tables from host to DRAM.

3) Check printhead temperature, if required, and configure printhead with firing pulse profile etc. accordingly. Instruct ISISlaves to also perform this operation.

4) Initiate printhead pre-heat sequence, if required. Instruct ISISlaves to also perform this operation

10.3.4 First page download

Buffer management in a SoPEC system is normally performed by the host.

1) The host communicates to the SoPEC CPU over the USB to check that DRAM space remaining is sufficient to download the first band.

2) The host downloads the first band (with the page header) to DRAM.

3) When the complete page header has been downloaded the SoPEC CPU processes the page header, calculates PEP register commands and write directly to PEP registers or to DRAM.

4) If PEP register commands have been written to DRAM, execute PEP commands from DRAM via PCU.

Poll ISISlaves for DRAM status and download compressed data to ISISlaves.

Remaining first page bands download and processing:

1) Check DRAM space remaining is sufficient to download the next band.

2) Download the next band with the band header to DRAM.

3) When the complete band header has been downloaded, process the band header according to whichever band-related register updating mechanism is being used.

Poll ISISlaves for DRAM status and download compressed data to ISISlaves.

10.3.5 Start printing

1) Wait until at least one band of the first page has been downloaded.

2) Start all the PEP Units by writing to their *Go* registers, via PCU commands executed from DRAM or direct CPU writes, in the suggested order defined in Table .

3) Print ready interrupt occurs (from PHI). Poll ISISlaves until print ready interrupt.

4) Start motor control (which may be on an ISISlave SoPEC), if first page, otherwise feed the next page. This step could occur before the print ready interrupt.

5) Drive LEDS, monitor paper status (which may be on an ISISlave SoPEC).

6) Wait for page alignment via page sensor(s) GPIO interrupt (which may be on an ISISlave SoPEC).

7) If the LineSyncMaster is a SoPEC its CPU instructs PHI to start producing master line syncs. Otherwise wait for an external device to produce line syncs.

8) Continue to download bands and process page and band headers for next page.

10.3.6 Next page(s) download

As for first page download, performed during printing of current page.

10.3.7 Between bands

When the finished band flags are asserted band related registers in the CDU, LBD and TE need to be re-programmed. This can be via PCU commands from DRAM. Typically only 3-5 commands per decompression unit need to be executed. These registers can also be reprogrammed directly by the

CPU or by updating from shadow registers. The finished band flag interrupts to the CPU, tell the CPU that the area of memory associated with the band is now free.

10.3.8 During page print

Typically during page printing ink usage is communicated to the QA chips.

1) Calculate ink printed (from PHI).

2) Decrement ink remaining (via QA chips).

3) Check amount of ink remaining (via QA chips). This operation may be better performed while the page is being printed rather than at the end of the page.

10.3.9 Page finish

These operations are typically performed when the page is finished:

1) Page finished interrupt occurs from PHI. Poll ISISlaves for page finished interrupts.

2) Shutdown the PEP blocks by de-asserting their Go registers in the suggested order in Table . This will set the PEP Unit state-machines to their startup states.

3) Communicate ink usage to QA chips, if required.

10.3.10 Start of next page

These operations are typically performed before printing the next page:

1) Re-program the PEP Units via PCU command processing from DRAM based on page header.

2) Go to Start printing.

10.3.11 End of document

1) Stop motor control. This may be on an ISISlave SoPEC.

10.3.12 Sleep mode

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block [16]. This may be as a result of a command from the host or as a result of a timeout.

1) Inform host PC of which parts of SoPEC system are about to sleep.

2) Instruct ISISlaves to enter sleep mode.

3) Store reusable cryptographic results in Power-Safe Storage (PSS).

4) Put ISIMaster SoPEC into defined sleep mode.

10.4 NORMAL OPERATION IN A MULTI-SOPEC SYSTEM - ISISLAVE SOPEC

This section the outline typical operation of an ISISlave SoPEC in a multi-SoPEC system. The ISIMaster can be another SoPEC or an ISI-Bridge chip. The ISISlave communicates with the host either via the ISIMaster or using a direct connection such as USB. For this use case we consider only an ISISlave that does not have a direct host connection. Buffer management in a SoPEC system is normally performed by the host.

10.4.1 Powerup

Powerup describes SoPEC initialisation following an external reset or the watchdog timer system reset.

A typical powerup sequence is:

1) Execute reset sequence for complete SoPEC.

2) CPU boot from ROM.

55

3) Basic configuration of CPU peripherals, SCB and DIU. DRAM initialisation.

4) Download and authentication of program (see Section 10.5.3).

5) Execution of program from DRAM.

6) Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.

7) SoPEC identification by sampling GPIO pins to determine ISIId. Communicate ISIId to ISIMaster.

8) Download and authenticate any further *datasets*.

10.4.2 ISI wakeup

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block [16]. Normally the CPU sub-system and the DRAM will be put in sleep mode but the SCB and power-safe storage (PSS) will still be enabled.

Wakeup describes SoPEC recovery from sleep mode with the SCB and power-safe storage (PSS) still enabled. In an ISISlave SoPEC, wakeup can be initiated following an ISI reset from the SCB.

A typical ISI wakeup sequence is:

1) Execute reset sequence for sections of SoPEC in sleep mode.

2) CPU boot from ROM, if CPU-subsystem was in sleep mode.

3) Basic configuration of CPU peripherals and DIU, and DRAM initialisation, if required.

4) Download and authentication of program using results in Power-Safe Storage (PSS) (see Section 10.5.3).

5) Execution of program from DRAM.

6) Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.

7) SoPEC identification by sampling GPIO pins to determine ISIId. Communicate ISIId to ISIMaster.

8) Download and authenticate any further *datasets*.

10.4.3 Print initialization

This sequence is typically performed at the start of a print job following powerup or wakeup:

1) Check amount of ink remaining via QA chips.

2) Download static data e.g. dither matrices, dead nozzle tables from ISI to DRAM.

3) Check printhead temperature, if required, and configure printhead with firing pulse profile etc. accordingly.

4) Initiate printhead pre-heat sequence, if required.

10.4.4 First page download

Buffer management in a SoPEC system is normally performed by the host via the ISI.

1) Check DRAM space remaining is sufficient to download the first band.

2) The host downloads the first band (with the page header) to DRAM via the ISI.

3) When the complete page header has been downloaded, process the page header, calculate PEP register commands and write directly to PEP registers or to DRAM.

4) If PEP register commands have been written to DRAM, execute PEP commands from DRAM via PCU.

Remaining first page bands download and processing:

1) Check DRAM space remaining is sufficient to download the next band.

2) The host downloads the first band (with the page header) to DRAM via the ISI.

3) When the complete band header has been downloaded, process the band header according to whichever band-related register updating mechanism is being used.

10.4.5 Start printing

1) Wait until at least one band of the first page has been downloaded.

2) Start all the PEP Units by writing to their *Go* registers, via PCU commands executed from DRAM or direct CPU writes, in the order defined in Table .

3) Print ready interrupt occurs (from PHI). Communicate to PrintMaster via ISI.

4) Start motor control, if attached to this ISISlave, when requested by PrintMaster, if first page, otherwise feed next page. This step could occur before the print ready interrupt

5) Drive LEDS, monitor paper status, if on this ISISlave SoPEC, when requested by PrintMaster

6) Wait for page alignment via page sensor(s) GPIO interrupt, if on this ISISlave SoPEC, and send to PrintMaster.

7) Wait for line sync and commence printing.

8) Continue to download bands and process page and band headers for next page.

10.4.6 Next page(s) download

As for first band download, performed during printing of current page.

10.4.7 Between bands

When the finished band flags are asserted band related registers in the CDU, LBD and TE need to be re-programmed. This can be via PCU commands from DRAM. Typically only 3-5 commands per decompression unit need to be executed. These registers can also be reprogrammed directly by the CPU or by updating from shadow registers. The finished band flag interrupts to the CPU tell the CPU that the area of memory associated with the band is now free.

10.4.8 During page print

Typically during page printing ink usage is communicated to the QA chips.

1) Calculate ink printed (from PHI).

2) Decrement ink remaining (via QA chips).

3) Check amount of ink remaining (via QA chips). This operation may be better performed while the page is being printed rather than at the end of the page.

10.4.9 Page finish

These operations are typically performed when the page is finished:

1) Page finished interrupt occurs from PHI. Communicate page finished interrupt to PrintMaster.

2) Shutdown the PEP blocks by de-asserting their Go registers in the suggested order in Table . This will set the PEP Unit state-machines to their startup states.

3) Communicate ink usage to QA chips, if required.

10.4.10 Start of next page

These operations are typically performed before printing the next page:

1) Re-program the PEP Units via PCU command processing from DRAM based on page header.

2) Go to Start printing.

10.4.11 End of document

Stop motor control, if attached to this ISISlave, when requested by PrintMaster.

10.4.12 Powerdown

In this mode SoPEC is no longer powered.

1) Powerdown ISISlave SoPEC when instructed by ISIMaster.

10.4.13 Sleep

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block [16]. This may be as a result of a command from the host or ISIMaster or as a result of a timeout.

1) Store reusable cryptographic results in Power-Safe Storage (PSS).

2) Put SoPEC into defined sleep mode.

10.5 SECURITY USE CASES

Please see the 'SoPEC Security Overview' [9] document for a more complete description of SoPEC security issues. The SoPEC boot operation is described in the ROM chapter of the SoPEC hardware design specification, Section 17.2.

10.5.1 Communication with the QA chips

Communication between SoPEC and the QA chips (i.e. INK_QA and PRINTER_QA) will take place on at least a per power cycle and per page basis. Communication with the QA chips has three principal purposes: validating the presence of genuine QA chips (i.e the printer is using approved consumables), validation of the amount of ink remaining in the cartridge and authenticating the operating parameters for the printer. After each page has been printed, SoPEC is expected to communicate the number of dots fired per ink plane to the QA chipset. SoPEC may also initiate decoy communications with the QA chips from time to time.

Process:

- When validating ink consumption SoPEC is expected to principally act as a conduit between the PRINTER_QA and INK_QA chips and to take certain actions (basically enable or disable printing and report status to host PC) based on the result. The communication channels are insecure but all traffic is signed to guarantee authenticity.

Known Weaknesses

- All communication to the QA chips is over the LSS interfaces using a serial communication protocol. This is open to observation and so the communication protocol could be reverse engineered. In this case both the PRINTER_QA and INK_QA chips could be replaced by impostor devices (e.g. a single FPGA) that successfully emulated the communication protocol. As this would require physical modification of each printer this is considered to be an acceptably low risk. Any messages that are not signed by one of the symmetric keys (such as the SoPEC_id_key) could be reverse engineered. The imposter device must also have access to the appropriate keys to crack the system.
- If the secret keys in the QA chips are exposed or cracked then the system, or parts of it, is compromised.

Assumptions:

[1] The QA chips are not involved in the authentication of downloaded SoPEC code

[2] The QA chip in the ink cartridge (INK_QA) does not directly affect the operation of the cartridge in any way i.e. it does not inhibit the flow of ink etc.

[3] The INK_QA and PRINTER_QA chips are identical in their virgin state. They only become a INK_QA or PRINTER_QA after their FlashROM has been programmed.

10.5.2   Authentication of downloaded code in a single SoPEC system

Process:

1)   SoPEC identification by activity on USB end-points 2-4 indicates it is the ISIMaster (unless the SoPEC CPU has explicitly disabled this function).

2)   The program is downloaded to the embedded DRAM.

3)   The CPU calculates a SHA-1 hash digest of the downloaded program.

4)   The *ResetSrc* register in the CPR block is read to determine whether or not a power-on reset occurred.

5)   If a power-on reset occurred the signature of the downloaded code (which needs to be in a known location such as the first or last N bytes of the downloaded code) is decrypted using the Silverbrook public boot0key stored in ROM. This decrypted signature is the expected SHA-1 hash of the accompanying program. The encryption algorithm is likely to be a public key algorithm such as RSA. If a power-on reset did not occur then the expected SHA-1 hash is retrieved from the PSS and the compute intensive decryption is not required.

6)   The calculated and expected hash values are compared and if they match then the programs authenticity has been verified.

7)   If the hash values do not match then the host PC is notified of the failure and the SoPEC will await a new program download.

8)   If the hash values match then the CPU starts executing the downloaded program.

9)   If, as is very likely, the downloaded program wishes to download subsequent programs (such as OEM code) it is responsible for ensuring the authenticity of everything it downloads. The downloaded program may contain public keys that are used to authenticate subsequent downloads, thus forming a hierarchy of authentication. The SoPEC ROM does not control these authentications - it is solely concerned with verifying that the first program downloaded has come from a trusted source.

10)   At some subsequent point OEM code starts executing. The Silverbrook supervisor code acts as an O/S to the OEM user mode code. The OEM code must access most SoPEC functionality via system calls to the Silverbrook code.

11)   The OEM code is expected to perform some simple 'turn on the lights' tasks after which the host PC is informed that the printer is ready to print and the *Start Printing* use case comes into play.

Known Weaknesses:

•   If the Silverbrook private boot0key is exposed or cracked then the system is seriously compromised. A ROM mask change would be required to reprogram the boot0key.

59

10.5.3   Authentication of downloaded code in a multi-SoPEC system

10.5.3.1 ISIMaster SoPEC Process:

1)   SoPEC identification by activity on USB end-points 2-4 indicates it is the ISIMaster.

2)   The SCB is configured to broadcast the data received from the host PC.

3)   The program is downloaded to the embedded DRAM and broadcasted to all ISISlave SoPECs over the ISI.

4)   The CPU calculates a SHA-1 hash digest of the downloaded program.

5)   The *ResetSrc* register in the CPR block is read to determine whether or not a power-on reset occurred.

6)   If a power-on reset occurred the signature of the downloaded code (which needs to be in a known location such as the first or last N bytes of the downloaded code) is decrypted using the Silverbrook public boot0key stored in ROM. This decrypted signature is the expected SHA-1 hash of the accompanying program. The encryption algorithm is likely to be a public key algorithm such as RSA. If a power-on reset did not occur then the expected SHA-1 hash is retrieved from the PSS and the compute intensive decryption is not required.

7)   The calculated and expected hash values are compared and if they match then the programs authenticity has been verified.

8)   If the hash values do not match then the host PC is notified of the failure and the SoPEC will await a new program download.

9)   If the hash values match then the CPU starts executing the downloaded program.

10)   It is likely that the downloaded program will poll each ISISlave SoPEC for the result of its authentication process and to determine the number of slaves present and their ISIIds.

11)   If any ISISlave SoPEC reports a failed authentication then the ISIMaster communicates this to the host PC and the SoPEC will await a new program download.

12)   If all ISISlaves report successful authentication then the downloaded program is responsible for the downloading, authentication and distribution of subsequent programs within the multi-SoPEC system.

13)   At some subsequent point OEM code starts executing. The Silverbrook supervisor code acts as an O/S to the OEM user mode code. The OEM code must access most SoPEC functionality via system calls to the Silverbrook code.

14)   The OEM code is expected to perform some simple 'turn on the lights' tasks after which the master SoPEC determines that all SoPECs are ready to print. The host PC is informed that the printer is ready to print and the *Start Printing* use case comes into play.

10.5.3.2 ISISlave SoPEC Process:

1)   When the CPU comes out of reset the SCB will be in slave mode, and the SCB is already configured to receive data from both the ISI and USB.

2)   The program is downloaded (via ISI or USB) to embedded DRAM.

3)   The CPU calculates a SHA-1 hash digest of the downloaded program.

4) The *ResetSrc* register in the CPR block is read to determine whether or not a power-on reset occurred.

5) If a power-on reset occurred the signature of the downloaded code (which needs to be in a known location such as the first or last N bytes of the downloaded code) is decrypted using the Silverbrook public boot0key stored in ROM. This decrypted signature is the expected SHA-1 hash of the accompanying program. The encryption algorithm is likely to be a public key algorithm such as RSA. If a power-on reset did not occur then the expected SHA-1 hash is retrieved from the PSS and the compute intensive decryption is not required.

6) The calculated and expected hash values are compared and if they match then the programs authenticity has been verified.

7) If the hash values do not match, then the ISISlave device will await a new program again

8) If the hash values match then the CPU starts executing the downloaded program.

9) It is likely that the downloaded program will communicate the result of its authentication process to the ISIMaster. The downloaded program is responsible for determining the SoPECs ISIId, receiving and authenticating any subsequent programs.

10) At some subsequent point OEM code starts executing. The Silverbrook supervisor code acts as an O/S to the OEM user mode code. The OEM code must access most SoPEC functionality via system calls to the Silverbrook code.

11) The OEM code is expected to perform some simple 'turn on the lights' tasks after which the master SoPEC is informed that this slave is ready to print. The *Start Printing* use case then comes into play.

Known Weaknesses

- If the Silverbrook private boot0key is exposed or cracked then the system is seriously compromised.

- ISI is an open interface i.e. messages sent over the ISI are in the clear. The communication channels are insecure but all traffic is signed to guarantee authenticity. As all communication over the ISI is controlled by Supervisor code on both the ISIMaster and ISISlave then this also provides some protection against software attacks.

10.5.4   Authentication and upgrade of operating parameters for a printer

The SoPEC IC will be used in a range of printers with different capabilities (e.g. A3/A4 printing, printing speed, resolution etc.). It is expected that some printers will also have a software upgrade capability which would allow a user to purchase a license that enables an upgrade in their printer's capabilities (such as print speed). To facilitate this it must be possible to securely store the operating parameters in the PRINTER_QA chip, to securely communicate these parameters to the SoPEC and to securely reprogram the parameters in the event of an upgrade. Note that each printing SoPEC (as opposed to a SoPEC that is only used for the storage of data) will have its own PRINTER_QA chip (or at least access to a PRINTER_QA that contains the SoPEC's SoPEC_id_key). Therefore both ISIMaster and ISISlave SoPECs will need to authenticate operating parameters.

Process:

1) Program code is downloaded and authenticated as described in sections 10.5.2 and 10.5.3 above.

2) The program code has a function to create the SoPEC_id_key from the unique SoPEC_id that was programmed when the SoPEC was manufactured.

3) The SoPEC retrieves the signed operating parameters from its PRINTER_QA chip. The PRINTER_QA chip uses the SoPEC_id_key (which is stored as part of the pairing process executed during printhead assembly manufacture & test) to sign the operating parameters which are appended with a random number to thwart replay attacks.

4) The SoPEC checks the signature of the operating parameters using its SoPEC_id_key. If this signature authentication process is successful then the operating parameters are considered valid and the overall boot process continues. If not the error is reported to the host PC.

5) Operating parameters may also be set or upgraded using a second key, the *PrintEngineLicense_key*, which is stored on the PRINTER_QA and used to authenticate the change in operating parameters.

Known Weaknesses:

• It may be possible to retrieve the unique SoPEC_id by placing the SoPEC in test mode and scanning it out. It is certainly possible to obtain it by reverse engineering the device. Either way the SoPEC_id (and by extension the SoPEC_id_key) so obtained is valid only for that specific SoPEC and so printers may only be compromised one at a time by parties with the appropriate specialised equipment. Furthermore even if the SoPEC_id is compromised, the other keys in the system, which protect the authentication of consumables and of program code, are unaffected.

10.6 MISCELLANEOUS USE CASES

There are many miscellaneous use cases such as the following examples. Software running on the SoPEC CPU or host will decide on what actions to take in these scenarios.

10.6.1 Disconnect / Re-connect of QA chips.

1) Disconnect of a QA chip between documents or if ink runs out mid-document.

2) Re-connect of a QA chip once authenticated e.g. ink cartridge replacement should allow the system to resume and print the next document

10.6.2 Page arrives before print ready interrupt.

1) Engage clutch to stop paper until print ready interrupt occurs.

10.6.3 Dead-nozzle table upgrade

This sequence is typically performed when dead nozzle information needs to be updated by performing a printhead dead nozzle test.

1) Run printhead nozzle test sequence

2) Either host or SoPEC CPU converts dead nozzle information into dead nozzle table.

3) Store dead nozzle table on host.

4) Write dead nozzle table to SoPEC DRAM.

10.7 FAILURE MODE USE CASES

10.7.1 System errors and security violations

System errors and security violations are reported to the SoPEC CPU and host. Software running on the SoPEC CPU or host will then decide what actions to take.

Silverbrook code authentication failure.

1) Notify host PC of authentication failure.

2) Abort print run.

OEM code authentication failure.

1) Notify host PC of authentication failure.

2) Abort print run.

Invalid QA chip(s).

1) Report to host PC.

2) Abort print run.

MMU security violation interrupt.

1) This is handled by exception handler.

2) Report to host PC

3) Abort print run.

Invalid address interrupt from PCU.

1) This is handled by exception handler.

2) Report to host PC.

3) Abort print run.

Watchdog timer interrupt.

1) This is handled by exception handler.

2) Report to host PC.

3) Abort print run.

Host PC does not acknowledge message that SoPEC is about to power down.

1) Power down anyway.

10.7.2 Printing errors

Printing errors are reported to the SoPEC CPU and host. Software running on the host or SoPEC CPU will then decide what actions to take.

Insufficient space available in SoPEC compressed band-store to download a band.

1) Report to the host PC.

Insufficient ink to print.

1) Report to host PC.

Page not downloaded in time while printing.

1) Buffer underrun interrupt will occur.

2) Report to host PC and abort print run.

JPEG decoder error interrupt.

1) Report to host PC.

CPU SUBSYSTEM

11 Central Processing Unit (CPU)

11.1 OVERVIEW

The CPU block consists of the CPU core, MMU, cache and associated logic. The principal tasks for the program running on the CPU to fulfill in the system are:

Communications:

- Control the flow of data from the USB interface to the DRAM and ISI
- Communication with the host via USB or ISI
- Running the USB device driver

PEP Subsystem Control:

- Page and band header processing (may possibly be performed on host PC)
- Configure printing options on a per band, per page, per job or per power cycle basis
- Initiate page printing operation in the PEP subsystem
- Retrieve dead nozzle information from the printhead interface (PHI) and forward to the host PC
- Select the appropriate firing pulse profile from a set of predefined profiles based on the printhead characteristics
- Retrieve printhead temperature via the PHI

Security:

- Authenticate downloaded program code
- Authenticate printer operating parameters
- Authenticate consumables via the PRINTER_QA and INK_QA chips
- Monitor ink usage
- Isolation of OEM code from direct access to the system resources

Other:

- Drive the printer motors using the GPIO pins
- Monitoring the status of the printer (paper jam, tray empty etc.)
- Driving front panel LEDs
- Perform post-boot initialisation of the SoPEC device
- Memory management (likely to be in conjunction with the host PC)
- Miscellaneous housekeeping tasks

To control the Print Engine Pipeline the CPU is required to provide a level of performance at least equivalent to a 16-bit Hitachi H8-3664 microcontroller running at 16 MHz. An as yet undetermined amount of additional CPU performance is needed to perform the other tasks, as well as to provide the potential for such activity as Netpage page assembly and processing, RIPing etc. The extra performance required is dominated by the signature verification task and the SCB (including the USB) management task. An operating system is not required at present. A number of CPU cores have been evaluated and the LEON P1754 is considered to be the most appropriate solution. A diagram of the CPU block is shown in Figure 15 below.

11.2 DEFINITIONS OF I/OS

Table 14. CPU Subsystem I/Os

64

Port name	Pins	I/O	Description
Clocks and Resets			
prst_n	1	In	Global reset. Synchronous to pclk, active low.
Pclk	1	In	Global clock
CPU to DIU DRAM interface			
cpu_adr[21:2]	20	Out	Address bus for both DRAM and peripheral access
cpu_dataout[31:0]	32	Out	Data out to both DRAM and peripheral devices. This should be driven at the same time as the *cpu_adr* and request signals.
dram_cpu_data[255:0]	256	In	Read data from the DRAM
cpu_diu_rreq	1	Out	Read request to the DIU DRAM
diu_cpu_rack	1	In	Acknowledge from DIU that read request has been accepted.
diu_cpu_rvalid	1	In	Signal from DIU telling SoPEC Unit that valid read data is on the *dram_cpu_data* bus
cpu_diu_wdatavalid	1	Out	Signal from the CPU to the DIU indicating that the data currently on the *cpu_diu_wdata* bus is valid and should be committed to the DIU posted write buffer
diu_cpu_write_rdy	1	In	Signal from the DIU indicating that the posted write buffer is empty
cpu_diu_wdadr[21:4]	18	Out	Write address bus to the DIU
cpu_diu_wdata[127:0]	128	Out	Write data bus to the DIU
cpu_diu_wmask[15:0]	16	Out	Write mask for the *cpu_diu_wdata* bus. Each bit corresponds to a byte of the 128-bit *cpu_diu_wdata* bus.
CPU to peripheral blocks			
cpu_rwn	1	Out	Common read/not-write signal from the CPU
cpu_acode[1:0]	2	Out	CPU access code signals. cpu_acode[0] - Program (0) / Data (1) access cpu_acode[1] - User (0) / Supervisor (1) access
cpu_cpr_sel	1	Out	CPR block select.
cpr_cpu_rdy	1	In	Ready signal to the CPU. When *cpr_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the CPR block and for a read cycle this means the data on *cpr_cpu_data* is valid.
cpr_cpu_berr	1	In	CPR bus error signal to the CPU.

cpr_cpu_data[31:0]	32	In	Read data bus from the CPR block
cpu_gpio_sel	1	Out	GPIO block select.
gpio_cpu_rdy	1	In	GPIO ready signal to the CPU.
gpio_cpu_berr	1	In	GPIO bus error signal to the CPU.
gpio_cpu_data[31:0]	32	In	Read data bus from the GPIO block
cpu_icu_sel	1	Out	ICU block select.
icu_cpu_rdy	1	In	ICU ready signal to the CPU.
icu_cpu_berr	1	In	ICU bus error signal to the CPU.
icu_cpu_data[31:0]	32	In	Read data bus from the ICU block
cpu_lss_sel	1	Out	LSS block select.
lss_cpu_rdy	1	In	LSS ready signal to the CPU.
lss_cpu_berr	1	In	LSS bus error signal to the CPU.
lss_cpu_data[31:0]	32	In	Read data bus from the LSS block
cpu_pcu_sel	1	Out	PCU block select.
pcu_cpu_rdy	1	In	PCU ready signal to the CPU.
pcu_cpu_berr	1	In	PCU bus error signal to the CPU.
pcu_cpu_data[31:0]	32	In	Read data bus from the PCU block
cpu_scb_sel	1	Out	SCB block select.
scb_cpu_rdy	1	In	SCB ready signal to the CPU.
scb_cpu_berr	1	In	SCB bus error signal to the CPU.
scb_cpu_data[31:0]	32	In	Read data bus from the SCB block
cpu_tim_sel	1	Out	Timers block select.
tim_cpu_rdy	1	In	Timers block ready signal to the CPU.
tim_cpu_berr	1	In	Timers bus error signal to the CPU.
tim_cpu_data[31:0]	32	In	Read data bus from the Timers block
cpu_rom_sel	1	Out	ROM block select.
rom_cpu_rdy	1	In	ROM block ready signal to the CPU.
rom_cpu_berr	1	In	ROM bus error signal to the CPU.
rom_cpu_data[31:0]	32	In	Read data bus from the ROM block
cpu_pss_sel	1	Out	PSS block select.
pss_cpu_rdy	1	In	PSS block ready signal to the CPU.
pss_cpu_berr	1	In	PSS bus error signal to the CPU.
pss_cpu_data[31:0]	32	In	Read data bus from the PSS block
cpu_diu_sel	1	Out	DIU register block select.
diu_cpu_rdy	1	In	DIU register block ready signal to the CPU.
diu_cpu_berr	1	In	DIU bus error signal to the CPU.
diu_cpu_data[31:0]	32	In	Read data bus from the DIU block
Interrupt signals			

icu_cpu_ilevel[3:0]	3	In	An interrupt is asserted by driving the appropriate priority level on *icu_cpu_ilevel*. These signals must remain asserted until the CPU executes an interrupt acknowledge cycle.
	3	Out	Indicates the level of the interrupt the CPU is acknowledging when *cpu_iack* is high
cpu_iack	1	Out	Interrupt acknowledge signal. The exact timing depends on the CPU core implementation
Debug signals			
diu_cpu_debug_valid	1	In	Signal indicating the data on the *diu_cpu_data* bus is valid debug data.
tim_cpu_debug_valid	1	In	Signal indicating the data on the *tim_cpu_data* bus is valid debug data.
scb_cpu_debug_valid	1	In	Signal indicating the data on the *scb_cpu_data* bus is valid debug data.
pcu_cpu_debug_valid	1	In	Signal indicating the data on the *pcu_cpu_data* bus is valid debug data.
lss_cpu_debug_valid	1	In	Signal indicating the data on the *lss_cpu_data* bus is valid debug data.
icu_cpu_debug_valid	1	In	Signal indicating the data on the *icu_cpu_data* bus is valid debug data.
gpio_cpu_debug_valid	1	In	Signal indicating the data on the *gpio_cpu_data* bus is valid debug data.
cpr_cpu_debug_valid	1	In	Signal indicating the data on the *cpr_cpu_data* bus is valid debug data.
debug_data_out	32	Out	Output debug data to be muxed on to the GPIO & PHI pins
debug_data_valid	1	Out	Debug valid signal indicating the validity of the data on *debug_data_out*. This signal is used in all debug configurations
debug_cntrl	33	Out	Control signal for each PHI bound debug data line indicating whether or not the debug data should be selected by the pin mux

11.3 REALTIME REQUIREMENTS

The SoPEC realtime requirements have yet to be fully determined but they may be split into three categories: hard, firm and soft

11.3.1 Hard realtime requirements

Hard requirements are tasks that must be completed before a certain deadline or failure to do so will result in an error perceptible to the user (printing stops or functions incorrectly). There are three hard realtime tasks:

- Motor control: The motors which feed the paper through the printer at a constant speed during printing are driven directly by the SoPEC device. Four periodic signals with different phase relationships need to be generated to ensure the paper travels smoothly through the printer. The generation of these signals is handled by the GPIO hardware (see section 13.2 for more details) but the CPU is responsible for enabling these signals (i.e. to start or stop the motors) and coordinating the movement of the paper with the printing operation of the printhead.

- Buffer management: Data enters the SoPEC via the SCB at an uneven rate and is consumed by the PEP subsystem at a different rate. The CPU is responsible for managing the DRAM buffers to ensure that neither overrun nor underrun occur. This buffer management is likely to be performed under the direction of the host.

- Band processing: In certain cases PEP registers may need to be updated between bands. As the timing requirements are most likely too stringent to be met by direct CPU writes to the PCU a more likely scenario is that a set of shadow registers will programmed in the compressed page units before the current band is finished, copied to band related registers by the finished band signals and the processing of the next band will continue immediately. An alternative solution is that the CPU will construct a DRAM based set of commands (see section 21.8.5 for more details) that can be executed by the PCU. The task for the CPU here is to parse the band headers stored in DRAM and generate a DRAM based set of commands for the next number of bands. The location of the DRAM based set of commands must then be written to the PCU before the current band has been processed by the PEP subsystem. It is also conceivable (but currently considered unlikely) that the host PC could create the DRAM based commands. In this case the CPU will only be required to point the PCU to the correct location in DRAM to execute commands from.

11.3.2   Firm requirements

Firm requirements are tasks that should be completed by a certain time or failure to do so will result in a degradation of performance but not an error. The majority of the CPU tasks for SoPEC fall into this category including all interactions with the QA chips, program authentication, page feeding, configuring PEP registers for a page or job, determining the firing pulse profile, communication of printer status to the host over the USB and the monitoring of ink usage. The authentication of downloaded programs and messages will be the most compute intensive operation the CPU will be required to perform. Initial investigations indicate that the LEON processor, running at 160 MHz, will easily perform three authentications in under a second.

Table 15. Expected firm requirements

Requirement	Duration
Power-on to start of printing first page [USB and slave SoPEC	~ 8 secs ??

enumeration, 3 or more RSA signature verifications, code and compressed page data download and chip initialisation]	
Wake-up from sleep mode to start printing [3 or more SHA-1 / RSA operations, code and compressed page data download and chip re-initialisation	~ 2 secs
Authenticate ink usage in the printer	~ 0.5 secs
Determining firing pulse profile	~ 0.1 secs
Page feeding, gap between pages	OEM dependent
Communication of printer status to host PC	~ 10 ms
Configuring PEP registers	??

11.3.3 Soft requirements

Soft requirements are tasks that need to be done but there are only light time constraints on when they need to be done. These tasks are performed by the CPU when there are no pending higher priority tasks. As the SoPEC CPU is expected to be lightly loaded these tasks will mostly be executed soon after they are scheduled.

11.4 BUS PROTOCOLS

As can be seen from Figure 15 above there are different buses in the CPU block and different protocols are used for each bus. There are three buses in operation:

11.4.1 AHB bus

The LEON CPU core uses an AMBA2.0 AHB bus to communicate with memory and peripherals (usually via an APB bridge). See the AMBA specification [38], section 5 of the LEON users manual [37] and section 11.6.6.1 of this document for more details.

11.4.2 CPU to DIU bus

This bus conforms to the DIU bus protocol described in Section 20.14.8. Note that the address bus used for DIU reads (i.e. *cpu_adr(21:2)*) is also that used for CPU subsystem with bus accesses while the write address bus (*cpu_diu_wadr*) and the read and write data buses (*dram_cpu_data* and *cpu_diu_wdata*) are private buses between the CPU and the DIU. The effective bus width differs between a read (256 bits) and a write (128 bits). As certain CPU instructions may require byte write access this will need to be supported by both the DRAM write buffer (in the AHB bridge) and the DIU. See section 11.6.6.1 for more details.

11.4.3 CPU Subsystem Bus

For access to the on-chip peripherals a simple bus protocol is used. The MMU must first determine which particular block is being addressed (and that the access is a valid one) so that the appropriate block select signal can be generated. During a write access CPU write data is driven out with the address and block select signals in the first cycle of an access. The addressed slave peripheral responds by asserting its ready signal indicating that it has registered the write data and the access can complete. The write data bus is common to all peripherals and is also used for CPU writes to the embedded DRAM. A read access is initiated by driving the address and select signals during the first cycle of an access. The addressed slave responds by placing the read data on its

bus and asserting its ready signal to indicate to the CPU that the read data is valid. Each block has a separate point-to-point data bus for read accesses to avoid the need for a tri-stateable bus.

All peripheral accesses are 32-bit (Programming note: *char* or *short* C types should not be used to access peripheral registers). The use of the ready signal allows the accesses to be of variable

5   length. In most cases accesses will complete in two cycles but three or four (or more) cycles accesses are likely for PEP blocks or IP blocks with a different native bus interface. All PEP blocks are accessed via the PCU which acts as a bridge. The PCU bus uses a similar protocol to the CPU subsystem bus but with the PCU as the bus master.

The duration of accesses to the PEP blocks is influenced by whether or not the PCU is executing

10   commands from DRAM. As these commands are essentially register writes the CPU access will need to wait until the PCU bus becomes available when a register access has been completed. This could lead to the CPU being stalled for up to 4 cycles if it attempts to access PEP blocks while the PCU is executing a command. The size and probability of this penalty is sufficiently small to have any significant impact on performance.

15   In order to support user mode (i.e. OEM code) access to certain peripherals the CPU subsystem bus propagates the CPU function code signals (*cpu_acode[1:0]*). These signals indicate the type of address space (i.e. User/Supervisor and Program/Data) being accessed by the CPU for each access. Each peripheral must determine whether or not the CPU is in the correct mode to be granted access to its registers and in some cases (e.g. Timers and GPIO blocks) different access

20   permissions can apply to different registers within the block. If the CPU is not in the correct mode then the violation is flagged by asserting the block's bus error signal (*block_cpu_berr*) with the same timing as its ready signal (*block_cpu_rdy*) which remains deasserted. When this occurs invalid read accesses should return 0 and write accesses should have no effect.

Figure 16 shows two examples of the peripheral bus protocol in action. A write to the LSS block

25   from code running in supervisor mode is successfully completed. This is immediately followed by a read from a PEP block via the PCU from code running in user mode. As this type of access is not permitted the access is terminated with a bus error. The bus error exception processing then starts directly after this - no further accesses to the peripheral should be required as the exception handler should be located in the DRAM.

30   Each peripheral acts as a slave on the CPU subsystem bus and its behavior is described by the state machine in section 11.4.3.1

11.4.3.1   CPU subsystem bus slave state machine

CPU subsystem bus slave operation is described by the state machine in Figure 17.This state machine will be implemented in each CPU subsystem bus slave. The only new signals mentioned

35   here are the *valid_access* and *reg_available* signals. The *valid_access* is determined by comparing the *cpu_acode* value with the block or register (in the case of a block that allow user access on a per register basis such as the GPIO block) access permissions and asserting *valid_access* if the permissions agree with the CPU mode. The *reg_available* signal is only required in the PCU or in blocks that are not capable of two-cycle access (e.g. blocks containing imported IP with different

40   bus protocols). In these blocks the *reg_available* signal is an internal signal used to insert wait

states (by delaying the assertion of *block_cpu_rdy*) until the CPU bus slave interface can gain access to the register.

When reading from a register that is less than 32 bits wide the CPU subsystems bus slave should return zeroes on the unused upper bits of the *block_cpu_data* bus.

5   To support debug mode the contents of the register selected for debug observation, *debug_reg*, are always output on the *block_cpu_data* bus whenever a read access is not taking place. See section 11.8 for more details of debug operation.

11.5   LEON CPU

The LEON processor is an open-source implementation of the IEEE-1754 standard (SPARC V8)

10   instruction set. LEON is available from and actively supported by Gaisler Research (www.gaisler.com).

The following features of the LEON-2 processor will be utilised on SoPEC:

- IEEE-1754 (SPARC V8) compatible integer unit with 5-stage pipeline
15   - Separate instruction and data cache (Harvard architecture). 1 kbyte direct mapped caches will be used for both.
- Full implementation of AMBA-2.0 AHB on-chip bus

The standard release of LEON incorporates a number of peripherals and support blocks which will not be included on SoPEC. The LEON core as used on SoPEC will consist of: 1) the LEON integer

20   unit, 2) the instruction and data caches (currently 1kB each), 3) the cache control logic, 4) the AHB interface and 5) possibly the AHB controller (although this functionality may be implemented in the LEON AHB bridge).

The version of the LEON database that the SoPEC LEON components will be sourced from is LEON2-1.0.7 although later versions may be used if they offer worthwhile functionality or bug fixes

25   that affect the SoPEC design.

The LEON core will be clocked using the system clock, *pclk*, and reset using the *prst_n_section[1]* signal. The ICU will assert all the hardware interrupts using the protocol described in section 11.9. The LEON hardware multipliers and floating-point unit are not required. SoPEC will use the recommended 8 register window configuration.

30   Further details of the SPARC V8 instruction set and the LEON processor can be found in [36] and [37] respectively.

11.5.1   LEON Registers

Only two of the registers described in the LEON manual are implemented on SoPEC - the LEON configuration register and the Cache Control Register (CCR). The addresses of these registers are

35   shown in Table 16. The configuration register bit fields are described below and the CCR is described in section 11.7.1.1.

11.5.1.1 LEON configuration register

The LEON configuration register allows runtime software to determine the settings of LEONs various configuration options. This is a read-only register whose value for the SoPEC ASIC will be

0x1071_8C00. Further descriptions of many of the bitfields can be found in the LEON manual. The values used for SoPEC are highlighted in bold for clarity.

Table 16. LEON Configuration Register

Field Name	bit(s)	Description
WriteProtection	1:0	Write protection type. 00 - none 01 - standard
PCICore	3:2	PCI core type 00 - none · 01 - InSilicon 10 - ESA 11 - Other
FPUType	5:4	FPU type. 00 - none 01 - Meiko
MemStatus	6	0 - No memory status and failing address register present 1 - Memory status and failing address register present
Watchdog	7	0 - Watchdog timer not present (Note this refers to the LEON watchdog timer in the LEON timer block). 1 - Watchdog timer present
UMUL/SMUL	8	0 - UMUL/SMUL instructions are not implemented 1 - UMUL/SMUL instructions are implemented
UDIV/SDIV	9	0 - UMUL/SMUL instructions are not implemented 1 - UMUL/SMUL instructions are implemented
DLSZ	11:10	Data cache line size in 32-bit words: 00 - 1 word 01 - 2 words 10 - 4 words 11 - 8 words
DCSZ	14:12	Data cache size in kBbytes = 2^{DCSZ}. SoPEC DCSZ = 0.
ILSZ	16:15	Instruction cache line size in 32-bit words: 00 - 1 word 01 - 2 words 10 - 4 words 11 - 8 words
ICSZ	19:17	Instruction cache size in kBbytes = 2^{ICSZ}. SoPEC ICSZ = 0.
RegWin	24:20	The implemented number of SPARC register windows - 1. SoPEC value = 7.

UMAC/SMAC	25	0 - UMAC/SMAC instructions are not implemented 1 - UMAC/SMAC instructions are implemented
Watchpoints	28:26	The implemented number of hardware watchpoints. SoPEC value = 4.
SDRAM	29	0 - SDRAM controller not present 1 - SDRAM controller present
DSU	30	0 - Debug Support Unit not present 1 - Debug Support Unit present
Reserved	31	Reserved. SoPEC value = 0.

11.6   MEMORY MANAGEMENT UNIT (MMU)

Memory Management Units are typically used to protect certain regions of memory from invalid accesses, to perform address translation for a virtual memory system and to maintain memory page status (swapped-in, swapped-out or unmapped)

The SoPEC MMU is a much simpler affair whose function is to ensure that all regions of the SoPEC memory map are adequately protected. The MMU does not support virtual memory and physical addresses are used at all times. The SoPEC MMU supports a full 32-bit address space. The SoPEC memory map is depicted in Figure 18 below.

The MMU selects the relevant bus protocol and generates the appropriate control signals depending on the area of memory being accessed. The MMU is responsible for performing the address decode and generation of the appropriate block select signal as well as the selection of the correct block read bus during a read access. The MMU will need to support all of the bus transactions the CPU can produce including interrupt acknowledge cycles, aborted transactions etc.

When an MMU error occurs (such as an attempt to access a supervisor mode only region when in user mode) a bus error is generated. While the LEON can recognise different types of bus error (e.g. data store error, instruction access error) it handles them in the same manner as it handles all traps i.e it will transfer control to a trap handler. No extra state information is be stored because of the nature of the trap. The location of the trap handler is contained in the TBR (Trap Base Register). This is the same mechanism as is used to handle interrupts.

11.6.1   CPU-bus peripherals address map

The address mapping for the peripherals attached to the CPU-bus is shown in Table 17 below. The MMU performs the decode of the high order bits to generate the relevant cpu_block_select signal. Apart from the PCU, which decodes the address space for the PEP blocks, each block only needs to decode as many bits of cpu_adr[11:2] as required to address all the registers within the block.

Table 17. CPU-bus peripherals address map

Block_base	Address
ROM_base	0x0000_0000
MMU_base	0x0001_0000

TIM_base	0x0001_1000
LSS_base	0x0001_2000
GPIO_base	0x0001_3000
SCB_base	0x0001_4000
ICU_base	0x0001_5000
CPR_base	0x0001_6000
DIU_base	0x0001_7000
PSS_base	0x0001_8000
Reserved	0x0001_9000 to 0x0001_FFFF
PCU_base	0x0002_0000

11.6.2 DRAM Region Mapping

The embedded DRAM is broken into 8 regions, with each region defined by a lower and upper bound address and with its own access permissions.

The association of an area in the DRAM address space with a MMU region is completely under software control. Table 18 below gives one possible region mapping. Regions should be defined according to their access requirements and position in memory. Regions that share the same access requirements and that are contiguous in memory may be combined into a single region. The example below is purely for indicative purposes - real mappings are likely to differ significantly from this. Note that the RegionBottom and RegionTop fields in this example include the DRAM base address offset (0x4000_0000) which is not required when programming the *RegionNTop* and *RegionNBottom* registers. For more details, see 11.6.5.1 and 11.6.5.2.

Table 18. Example region mapping

Region	RegionBottom	RegionTop	Description
0	0x4000_0000	0x4000_0FFF	Silverbrook OS (supervisor) data
1	0x4000_1000	0x4000_BFFF	Silverbrook OS (supervisor) code
2	0x4000_C000	0x4000_C3FF	Silverbrook (supervisor/user) data
3	0x4000_C400	0x4000_CFFF	Silverbrook (supervisor/user) code
4	0x4026_D000	0x4026_D3FF	OEM (user) data
5	0x4026_D400	0x4026_DFFF	OEM (user) code
6	0x4027_E000	0x4027_FFFF	Shared Silverbrook/OEM space
7	0x4000_D000	0x4026_CFFF	Compressed page store (supervisor data)

11.6.3 Non-DRAM regions

As shown in Figure 18 the DRAM occupies only 2.5 MBytes of the total 4 GB SoPEC address space. The non-DRAM regions of SoPEC are handled by the MMU as follows:

ROM (0x0000_0000 to 0x0000_FFFF): The ROM block will control the access types allowed. The *cpu_acode[1:0]* signals will indicate the CPU mode and access type and the ROM block will assert *rom_cpu_berr* if an attempted access is forbidden. The protocol is described in more detail in

section 11.4.3. The ROM block access permissions are hard wired to allow all read accesses except to the *FuseChipID* registers which may only be read in supervisor mode.

MMU Internal Registers (0x0001_0000 to 0x0001_0FFF): The MMU is responsible for controlling the accesses to its own internal registers and will only allow data reads and writes (no instruction
5   fetches) from supervisor data space. All other accesses will result in the *mmu_cpu_berr* signal being asserted in accordance with the CPU native bus protocol.

CPU Subsystem Peripheral Registers (0x0001_1000 to 0x0001_FFFF): Each peripheral block will control the access types allowed. Every peripheral will allow supervisor data accesses (both read and write) and some blocks (e.g. Timers and GPIO) will also allow user data space accesses as
10   outlined in the relevant chapters of this specification. Neither supervisor nor user instruction fetch accesses are allowed to any block as it is not possible to execute code from peripheral registers. The bus protocol is described in section 11.4.3.

PCU Mapped Registers (0x0002_0000 to 0x0002_BFFF): All of the PEP blocks registers which are accessed by the CPU via the PCU will inherit the access permissions of the PCU. These access
15   permissions are hard wired to allow supervisor data accesses only and the protocol used is the same as for the CPU peripherals.

Unused address space (0x0002_C000 to 0x3FFF_FFFF and 0x4028_0000 to 0xFFFF_FFFF): All accesses to the unused portion of the address space will result in the *mmu_cpu_berr* signal being asserted in accordance with the CPU native bus protocol. These accesses will not propagate
20   outside of the MMU i.e. no external access will be initiated.

11.6.4   Reset exception vector and reference zero traps

When a reset occurs the LEON processor starts executing code from address 0x0000_0000. A common software bug is zero-referencing or null pointer de-referencing (where the program attempts to access the contents of address 0x0000_0000). To assist software debug the MMU will
25   assert a bus error every time the locations 0x0000_0000 to 0x0000_000F (i.e. the first 4 words of the reset trap) are accessed after the reset trap handler has legitimately been retrieved immediately after reset.

11.6.5   MMU Configuration Registers

The MMU configuration registers include the RDU configuration registers and two LEON registers.
30   Note that all the MMU configuration registers may only be accessed when the CPU is running in supervisor mode.

Table 19. MMU Configuration Registers

Address offset from MMU_base	Register	#bits	Reset	Description
0x00	Region0Bottom[21:5]	17	0x0_0000	This register contains the physical address that marks the bottom of region 0
0x04	Region0Top[21:5]	17	0xF_FFFF	This register contains the physical address that marks the top of region 0. Region 0 covers the

75

				entire address space after reset whereas all other regions are zero-sized initially.
0x08	Region1Bottom[21:5]	17	0xF_FFFF	This register contains the physical address that marks the bottom of region 1
0x0C	Region1Top[21:5]	17	0x0_0000	This register contains the physical address that marks the top of region 1
0x10	Region2Bottom[21:5]	17	0xF_FFFF	This register contains the physical address that marks the bottom of region 2
0x14	Region3Top[21:5]	17	0x0_0000	This register contains the physical address that marks the top of region 2
0x18	Region3Bottom[21:5]	17	0xF_FFFF	This register contains the physical address that marks the bottom of region 3
0x1C	Region3Top[21:5]	17	0x0_0000	This register contains the physical address that marks the top of region 3
0x20	Region4Bottom[21:5]	17	0xF_FFFF	This register contains the physical address that marks the bottom of region 4
0x24	Region4Top[21:5]	17	0x0_0000	This register contains the physical address that marks the top of region 4
0x28	Region5Bottom[21:5]	17	0xF_FFFF	This register contains the physical address that marks the bottom of region 5
0x2C	Region5Top[21:5]	17	0x0_0000	This register contains the physical address that marks the top of region 5
0x30	Region6Bottom[21:5]	17	0xF_FFFF	This register contains the physical address that marks the bottom of region 6
0x34	Region6Top[21:5]	17	0x0_0000	This register contains the physical address that marks the top of region 6
0x38	Region7Bottom[21:5]	17	0xF_FFFF	This register contains the physical address that marks the bottom of region 7
0x3C	Region7Top[21:5]	17	0x0_0000	This register contains the physical address that marks the top of region 7
0x40	Region0Control	6	0x07	Control register for region 0
0x44	Region1Control	6	0x07	Control register for region 1
0x48	Region2Control	6	0x07	Control register for region 2
0x4C	Region3Control	6	0x07	Control register for region 3
0x50	Region4Control	6	0x07	Control register for region 4
0x54	Region5Control	6	0x07	Control register for region 5
0x58	Region6Control	6	0x07	Control register for region 6
0x5C	Region7Control	6	0x07	Control register for region 7
0x60	RegionLock	8	0x00	Writing a 1 to a bit in the RegionLock register

				locks the value of the corresponding Region-Top, RegionBottom and RegionControl registers. The lock can only be cleared by a reset and any attempt to write to a locked register will result in a bus error.
0x64	BusTimeout	8	0xFF	This register should be set to the number of *pclk* cycles to wait after an access has started before aborting the access with a bus error. Writing 0 to this register disables the bus time-out feature.
0x68	ExceptionSource	6	0x00	This register identifies the source of the last exception. See Section 11.6.5.3 for details.
0x6C	DebugSelect	7	0x00	Contains address of the register selected for debug observation. It is expected that a number of pseudo-registers will be made available for debug observation and these will be outlined during the implementation phase.
0x80 to 0x108	RDU Registers			See Table for details.
0x140	LEON Configuration Register	32	0x1071_8 C00	The LEON configuration register is used by software to determine the configuration of this LEON implementation. See section 11.5.1.1 for details. This register is ReadOnly.
0x144	LEON Cache Control Register	32	0x0000_0 000	The LEON Cache Control Register is used to control the operation of the caches. See section 11.6 for details.

11.6.5.1 RegionTop and RegionBottom registers

The 20 Mbit of embedded DRAM on SoPEC is arranged as 81920 words of 256 bits each. All region boundaries need to align with a 256-bit word. Thus only 17 bits are required for the

5   *RegionNTop* and *RegionNBottom* registers. Note that the bottom 5 bits of the *RegionNTop* and *RegionNBottom* registers cannot be written to and read as '0' i.e. the *RegionNTop* and *RegionNBottom* registers represent byte-aligned DRAM addresses

Both the *RegionNTop* and *RegionNBottom* registers are inclusive i.e. the addresses in the registers are included in the region. Thus the size of a region is (*RegionNTop* - *RegionNBottom*) +1 DRAM

10   words.

If DRAM regions overlap (there is no reason for this to be the case but there is nothing to prohibit it either) then only accesses allowed by all overlapping regions are permitted. That is if a DRAM address appears in both Region1 and Region3 (for example) the *cpu_acode* of an access is

checked against the access permissions of both regions. If both regions permit the access then it will proceed but if either or both regions do not permit the access then it will not be allowed.

The MMU does not support negatively sized regions i.e. the value of the *RegionNTop* register should always be greater than or equal to the value of the *RegionNBottom* register. If *RegionNTop*

5   is lower in the address map than *RegionNTop* then the region is considered to be zero-sized and is ignored.

When both the *RegionNTop* and *RegionNBottom* registers for a region contain the same value the region is then simply one 256-bit word in length and this corresponds to the smallest possible active region.

10   *11.6.5.2   Region Control registers*

Each memory region has a control register associated with it. The *RegionNControl* register is used to set the access conditions for the memory region bounded by the *RegionNTop* and *RegionNBottom* registers. Table 20 describes the function of each bit field in the *RegionNControl* registers. All bits in a *RegionNControl* register are both readable and writable by design. However,

15   like all registers in the MMU, the *RegionNControl* registers can only be accessed by code running in supervisor mode.

Table 20. Region Control Register

Field Name	bit(s)	Description
SupervisorAccess	2:0	Denotes the type of access allowed when the CPU is running in Supervisor mode. For each access type a 1 indicates the access is permitted and a 0 indicates the access is not permitted. bit0 - Data read access permission bit1 - Data write access permission bit2 - Instruction fetch access permission
UserAccess	5:3	Denotes the type of access allowed when the CPU is running in User mode. For each access type a 1 indicates the access is permitted and a 0 indicates the access is not permitted. bit3 - Data read access permission bit4 - Data write access permission bit5 - Instruction fetch access permission

11.6.5.3   ExceptionSource Register

20   The SPARC V8 architecture allows for a number of types of memory access error to be trapped. These trap types and trap handling in general are described in chapter 7 of the SPARC architecture manual [36]. However on the LEON processor only *data_store_error* and *data_access_exception* trap types will result from an external (to LEON) bus error. According to the SPARC architecture manual the processor will automatically move to the next register window (i.e. it decrements the

25   current window pointer) and copies the program counters (PC and nPC) to two local registers in the

new window. The supervisor bit in the PSR is also set and the PSR can be saved to another local register by the trap handler (this does not happen automatically in hardware). The *ExceptionSource* register aids the trap handler by identifying the source of an exception. Each bit in the *ExceptionSource* register is set when the relevant trap condition and should be cleared by the trap handler by writing a '1' to that bit position.

Table 21. ExceptionSource Register

Field Name	bit(s)	Description
DramAccessExcptn	0	The permissions of an access did not match those of the DRAM region it was attempting to access. This bit will also be set if an attempt is made to access an undefined DRAM region (i.e. a location that is not within the bounds of any RegionTop/RegionBottom pair)
PeriAccessExcptn	1	An access violation occurred when accessing a CPU subsystem block. This occurs when the access permissions disagree with those set by the block.
UnusedAreaExcptn	2	An attempt was made to access an unused part of the memory map
LockedWriteExcptn	3	An attempt was made to write to a regions registers (RegionTop/Bottom/Control) after they had been locked.
ResetHandlerExcptn	4	An attempt was made to access a ROM location between 0x0000_0000 and 0x0000_000F after the reset handler was executed. The most likely cause of such an access is the use of an uninitialised pointer or structure.
TimeoutExcptn	5	A bus timeout condition occurred.

11.6.6   MMU Sub-block partition

As can be seen from Figure 19 and Figure 20 the MMU consists of three principal sub-blocks. For clarity the connections between these sub-blocks and other SoPEC blocks and between each of the sub-blocks are shown in two separate diagrams.

11.6.6.1 LEON AHB Bridge

The LEON AHB bridge consists of an AHB bridge to DIU and an AHB to CPU subsystem bus bridge. The AHB bridge will convert between the AHB and the DIU and CPU subsystem bus protocols but the address decoding and enabling of an access happens elsewhere in the MMU. The AHB bridge will always be a slave on the AHB. Note that the AMBA signals from the LEON core are contained within the ahbso and ahbsi records. The LEON records are described in more detail in section 11.7. Glue logic may be required to assist with enabling memory accesses, endianness coherency, interrupts and other miscellaneous signalling.

Table 22. LEON AHB bridge I/Os

Port name	Pins	I/O	Description
Global SoPEC signals			
prst_n	1	In	Global reset. Synchronous to *pclk*, active low.
pclk	1	In	Global clock
LEON core to LEON AHB signals (ahbsi and ahbso records)			
ahbsi.haddr[31:0]	32	In	AHB address bus
ahbsi.hwdata[31:0]	32	In	AHB write data bus
ahbso.hrdata[31:0]	32	Out	AHB read data bus
ahbsi.hsel	1	In	AHB slave select signal
ahbsi.hwrite	1	In	AHB write signal: 1 - Write access 0 - Read access
ahbsi.htrans	2	In	Indicates the type of the current transfer: 00 - IDLE 01 - BUSY 10 - NONSEQ 11 - SEQ
ahbsi.hsize	3	In	Indicates the size of the current transfer: 000 - Byte transfer 001 - Halfword transfer 010 - Word transfer 011 - 64-bit transfer (unsupported?) 1xx - Unsupported larger wordsizes
ahbsi.hburst	3	In	Indicates if the current transfer forms part of a burst and the type of burst: 000 - SINGLE 001 - INCR 010 - WRAP4 011 - INCR4 100 - WRAP8 101 - INCR8 110 - WRAP16 111 - INCR16
ahbsi.hprot	4	In	Protection control signals pertaining to the current access: hprot[0] - Opcode(0) / Data(1) access hprot[1] - User(0) / Supervisor access hprot[2] - Non-bufferable(0) / Bufferable(1) access (unsupported)

			hprot[3] - Non-cacheable(0) / Cacheable access
ahbsi.hmaster	4	In	Indicates the identity of the current bus master. This will always be the LEON core.
ahbsi.hmastlock	1	In	Indicates that the current master is performing a locked sequence of transfers.
ahbso.hready	1	Out	Active high ready signal indicating the access has completed
ahbso.hresp	2	Out	Indicates the status of the transfer: 00 - OKAY 01 - ERROR 10 - RETRY 11 - SPLIT
ahbso.hsplit[15:0]	16	Out	This 16-bit split bus is used by a slave to indicate to the arbiter which bus masters should be allowed attempt a split transaction. This feature will be unsupported on the AHB bridge
Toplevel/ Common LEON AHB bridge signals			
cpu_dataout[31:0]	32	Out	Data out bus to both DRAM and peripheral devices.
cpu_rwn	1	Out	Read/NotWrite signal. 1 = Current access is a read access, 0 = Current access is a write access
icu_cpu_ilevel[3:0]	4	In	An interrupt is asserted by driving the appropriate priority level on *icu_cpu_ilevel*. These signals must remain asserted until the CPU executes an interrupt acknowledge cycle.
cpu_icu_ilevel[3:0]	4	In	Indicates the level of the interrupt the CPU is acknowledging when *cpu_iack* is high
cpu_iack	1	Out	Interrupt acknowledge signal. The exact timing depends on the CPU core implementation
cpu_start_access	1	Out	Start Access signal indicating the start of a data transfer and that the *cpu_adr, cpu_dataout, cpu_rwn* and *cpu_acode* signals are all valid. This signal is only asserted during the first cycle of an access.
cpu_ben[1:0]	2	Out	Byte enable signals.
dram_cpu_data[255:0]	256	In	Read data from the DRAM.
diu_cpu_rreq	1	Out	Read request to the DIU.

diu_cpu_rack	1	In	Acknowledge from DIU that read request has been accepted.
diu_cpu_rvalid	1	In	Signal from DIU indicating that valid read data is on the *dram_cpu_data* bus
cpu_diu_wdatavalid	1	Out	Signal from the CPU to the DIU indicating that the data currently on the *cpu_diu_wdata* bus is valid and should be committed to the DIU posted write buffer
diu_cpu_write_rdy	1	In	Signal from the DIU indicating that the posted write buffer is empty
cpu_diu_wdadr[21:4]	18	Out	Write address bus to the DIU
cpu_diu_wdata[127:0]	128	Out	Write data bus to the DIU
cpu_diu_wmask[15:0]	16	Out	Write mask for the *cpu_diu_wdata* bus. Each bit corresponds to a byte of the 128-bit *cpu_diu_wdata* bus.
LEON AHB bridge to MMU Control Block signals			
cpu_mmu_adr	32	Out	CPU Address Bus.
mmu_cpu_data	32	In	Data bus from the MMU
mmu_cpu_rdy	1	In	Ready signal from the MMU
cpu_mmu_acode	2	Out	Access code signals to the MMU
mmu_cpu_berr	1	In	Bus error signal from the MMU
dram_access_en	1	In	DRAM access enable signal. A DRAM access cannot be initiated unless it has been enabled by the MMU control unit.

Description:

The LEON AHB bridge must ensure that all CPU bus transactions are functionally correct and that the timing requirements are met. The AHB bridge also implements a 128-bit DRAM write buffer to improve the efficiency of DRAM writes, particularly for multiple successive writes to DRAM. The AHB bridge is also responsible for ensuring endianness coherency i.e. guaranteeing that the correct data appears in the correct position on the data buses (*hrdata, cpu_dataout* and *cpu_mmu_wdata)* for every type of access. This is a requirement because the LEON uses big-endian addressing while the rest of SoPEC is little-endian.

The LEON AHB bridge will assert request signals to the DIU if the MMU control block deems the access to be a legal access. The validity (i.e. is the CPU running in the correct mode for the address space being accessed) of an access is determined by the contents of the relevant *RegionNControl* register. As the SPARC standard requires that all accesses are aligned to their word size (i.e. byte, half-word, word or double-word) and so it is not possible for an access to traverse a 256-bit boundary (as required by the DIU). Invalid DRAM accesses are not propagated to the DIU and will result in an error response (ahbso.hresp = '01') on the AHB. The DIU bus protocol

is described in more detail in section 20.9. The DIU will return a 256-bit dataword on *dram_cpu_data[255:0]* for every read access.

The CPU subsystem bus protocol is described in section 11.4.3. While the LEON AHB bridge performs the protocol translation between AHB and the CPU subsystem bus the select signals for each block are generated by address decoding in the CPU subsystem bus interface. The CPU subsystem bus interface also selects the correct read data bus, ready and error signals for the block being addressed and passes these to the LEON AHB bridge which puts them on the AHB bus. It is expected that some signals (especially those external to the CPU block) will need to be registered here to meet the timing requirements. Careful thought will be required to ensure that overall CPU access times are not excessively degraded by the use of too many register stages.

11.6.6.1.1   DRAM write buffer

The DRAM write buffer improves the efficiency of DRAM writes by aggregating a number of CPU write accesses into a single DIU write access. This is achieved by checking to see if a CPU write is to an address already in the write buffer and if so the write is immediately acknowledged (i.e. the *ahbsi.hready* signal is asserted without any wait states) and the DRAM write buffer updated accordingly. When the CPU write is to a DRAM address other than that in the write buffer then the current contents of the write buffer are sent to the DIU (where they are placed in the posted write buffer) and the DRAM write buffer is updated with the address and data of the CPU write. The DRAM write buffer consists of a 128-bit data buffer, an 18-bit write address tag and a 16-bit write mask. Each bit of the write mask indicates the validity of the corresponding byte of the write buffer as shown in Figure 21 below.

The operation of the DRAM write buffer is summarised by the following set of rules:

1) The DRAM write buffer only contains DRAM write data i.e. peripheral writes go directly to the addressed peripheral.

2) CPU writes to locations within the DRAM write buffer or to an empty write buffer (i.e. the write mask bits are all 0) complete with zero wait states regardless of the size of the write (byte/half-word/word/ double-word).

3) The contents of the DRAM write buffer are flushed to DRAM whenever a CPU write to a location outside the write buffer occurs, whenever a CPU read from a location within the write buffer occurs or whenever a write to a peripheral register occurs.

4) A flush resulting from a peripheral write will not cause any extra wait states to be inserted in the peripheral write access.

5) Flushes resulting from a DRAM accesses will cause wait states to be inserted until the DIU posted write buffer is empty. If the DIU posted write buffer is empty at the time the flush is required then no wait states will be inserted for a flush resulting from a CPU write or one wait state will be inserted for a flush resulting from a CPU read (this is to ensure that the DIU sees the write request ahead of the read request). Note that in this case further wait states will also be inserted as a result of the delay in servicing the read request by the DIU.

11.6.6.1.2   DIU interface waveforms

Figure 22 below depicts the operation of the AHB bridge over a sample sequence of DRAM transactions consisting of a read into the DCache, a double-word store to an address other than that currently in the DRAM write buffer followed by an ICache line refill. To avoid clutter a number of AHB control signals that are inputs to the MMU have been grouped together as ahbsi.CONTROL and only the ahbso.HREADY is shown of the output AHB control signals.

The first transaction is a single word load ('LD'). The MMU (specifically the MMU control block) uses the first cycle of every access (i.e. the address phase of an AHB transaction) to determine whether or not the access is a legal access. The read request to the DIU is then asserted in the following cycle (assuming the access is a valid one) and is acknowledged by the DIU a cycle later. Note that the time from *cpu_diu_rreq* being asserted and *diu_cpu_rack* being asserted is variable as it depends on the DIU configuration and access patterns of DIU requestors. The AHB bridge will insert wait states until it sees the *diu_cpu_rvalid* signal is high, indicating the data ('LD1') on the *dram_cpu_data* bus is valid. The AHB bridge terminates the read access in the same cycle by asserting the ahbso.HREADY signal (together with an 'OKAY' HRESP code). The AHB bridge also selects the appropriate 32 bits ('RD1') from the 256-bit DRAM line data ('LD1') returned by the DIU corresponding to the word address given by A1.

The second transaction is an AHB two-beat incrementing burst issued by the LEON acache block in response to the execution of a double-word store instruction. As LEON is a big endian processor the address issued ('A2') during the address phase of the first beat of this transaction is the address of the most significant word of the double-word while the address for the second beat ('A3') is that of the least significant word i.e. A3 = A2 +4. The presence of the DRAM write buffer allows these writes to complete without the insertion of any wait states. This is true even when, as shown here, the DRAM write buffer needs to be flushed into the DIU posted write buffer, provided the DIU posted write buffer is empty. If the DIU posted write buffer is not empty (as would be signified by *diu_cpu_write_rdy* being low) then wait states would be inserted until it became empty. The *cpu_diu_wdata* buffer builds up the data to be written to the DIU over a number of transactions ('BD1' and 'BD2' here) while the *cpu_diu_wmask* records every byte that has been written to since the last flush - in this case the lowest word and then the second lowest word are written to as a result of the double-word store operation.

The final transaction shown here is a DRAM read caused by an ICache miss. Note that the pipelined nature of the AHB bus allows the address phase of this transaction to overlap with the final data phase of the previous transaction. All ICache misses appear as single word loads ('LD') on the AHB bus. In this case we can see that the DIU is slower to respond to this read request than to the first read request because it is processing the write access caused by the DRAM write buffer flush. The ICache refill will complete just after the window shown in Figure 22.

11.6.6.2 CPU Subsystem Bus Interface

The CPU Subsystem Interface block handles all valid accesses to the peripheral blocks that comprise the CPU Subsystem.

Table 23. CPU Subsystem Bus Interface I/Os

Port name	Pins	I/O	Description
Global SoPEC signals			
prst_n	1	In	Global reset. Synchronous to *pclk*, active low.
pclk	1	In	Global clock
Toplevel/Common CPU Subsystem Bus Interface signals			
cpu_cpr_sel	1	Out	CPR block select.
cpu_gpio_sel	1	Out	GPIO block select.
cpu_icu_sel	1	Out	ICU block select.
cpu_lss_sel	1	Out	LSS block select.
cpu_pcu_sel	1	Out	PCU block select.
cpu_scb_sel	1	Out	SCB block select.
cpu_tim_sel	1	Out	Timers block select.
cpu_rom_sel	1	Out	ROM block select.
cpu_pss_sel	1	Out	PSS block select.
cpu_diu_sel	1	Out	DIU block select.
cpr_cpu_data[31:0]	32	In	Read data bus from the CPR block
gpio_cpu_data[31:0]	32	In	Read data bus from the GPIO block
icu_cpu_data[31:0]	32	In	Read data bus from the ICU block
lss_cpu_data[31:0]	32	In	Read data bus from the LSS block
pcu_cpu_data[31:0]	32	In	Read data bus from the PCU block
scb_cpu_data[31:0]	32	In	Read data bus from the SCB block
tim_cpu_data[31:0]	32	In	Read data bus from the Timers block
rom_cpu_data[31:0]	32	In	Read data bus from the ROM block
pss_cpu_data[31:0]	32	In	Read data bus from the PSS block
diu_cpu_data[31:0]	32	In	Read data bus from the DIU block
cpr_cpu_rdy	1	In	Ready signal to the CPU. When *cpr_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the CPR block and for a read cycle this means the data on *cpr_cpu_data* is valid.
gpio_cpu_rdy	1	In	GPIO ready signal to the CPU.
icu_cpu_rdy	1	In	ICU ready signal to the CPU.
lss_cpu_rdy	1	In	LSS ready signal to the CPU.
pcu_cpu_rdy	1	In	PCU ready signal to the CPU.
scb_cpu_rdy	1	In	SCB ready signal to the CPU.
tim_cpu_rdy	1	In	Timers block ready signal to the CPU.
rom_cpu_rdy	1	In	ROM block ready signal to the CPU.
pss_cpu_rdy	1	In	PSS block ready signal to the CPU.

diu_cpu_rdy	1	In	DIU register block ready signal to the CPU.
cpr_cpu_berr	1	In	Bus Error signal from the CPR block
gpio_cpu_berr	1	In	Bus Error signal from the GPIO block
icu_cpu_berr	1	In	Bus Error signal from the ICU block
lss_cpu_berr	1	In	Bus Error signal from the LSS block
pcu_cpu_berr	1	In	Bus Error signal from the PCU block
scb_cpu_berr	1	In	Bus Error signal from the SCB block
tim_cpu_berr	1	In	Bus Error signal from the Timers block
rom_cpu_berr	1	In	Bus Error signal from the ROM block
pss_cpu_berr	1	In	Bus Error signal from the PSS block
diu_cpu_berr	1	In	Bus Error signal from the DIU block
CPU Subsystem Bus Interface to MMU Control Block signals			
cpu_adr[19:12]	8	In	Toplevel CPU Address bus. Only bits 19-12 are required to decode the peripherals address space
peri_access_en	1	In	Enable Access signal. A peripheral access cannot be initiated unless it has been enabled by the MMU Control Unit
peri_mmu_data[31:0]	32	Out	Data bus from the selected peripheral
peri_mmu_rdy	1	Out	Data Ready signal. Indicates the data on the *peri_mmu_data* bus is valid for a read cycle or that the data was successfully written to the peripheral for a write cycle.
peri_mmu_berr	1	Out	Bus Error signal. Indicates a bus error has occurred in accessing the selected peripheral
CPU Subsystem Bus Interface to LEON AHB bridge signals			
cpu_start_access	1	In	Start Access signal from the LEON AHB bridge indicating the start of a data transfer and that the *cpu_adr, cpu_dataout, cpu_rwn* and *cpu_acode* signals are all valid. This signal is only asserted during the first cycle of an access.

Description:

The CPU Subsystem Bus Interface block performs simple address decoding to select a peripheral and multiplexing of the returned signals from the various peripheral blocks. The base addresses used for the decode operation are defined in Table . Note that access to the MMU configuration registers are handled by the MMU Control Block rather than the CPU Subsystem Bus Interface block. The CPU Subsystem Bus Interface block operation is described by the following pseudocode:

```
masked_cpu_adr = cpu_adr[17:12]
case (masked_cpu_adr)
```

86

```
        when TIM_base[17:12]
            cpu_tim_sel = peri_access_en      // The peri_access_en
    signal will have the
            peri_mmu_data = tim_cpu_data      // timing required for
    block selects
            peri_mmu_rdy = tim_cpu_rdy
            peri_mmu_berr = tim_cpu_berr
            all_other_selects = 0     //   Shorthand to ensure other
    cpu_block_sel signals
                                              //   remain deasserted
        when LSS_base[17:12]
            cpu_lss_sel = peri_access_en
            peri_mmu_data = lss_cpu_data
            peri_mmu_rdy = lss_cpu_rdy
            peri_mmu_berr = lss_cpu_berr
            all_other_selects = 0
        when GPIO_base[17:12]
            cpu_gpio_sel = peri_access_en
            peri_mmu_data = gpio_cpu_data
            peri_mmu_rdy = gpio_cpu_rdy
            peri_mmu_berr = gpio_cpu_berr
            all_other_selects = 0
        when SCB_base[17:12]
            cpu_scb_sel = peri_access_en
            peri_mmu_data = scb_cpu_data
            peri_mmu_rdy = scb_cpu_rdy
            peri_mmu_berr = scb_cpu_berr
            all_other_selects = 0
        when ICU_base[17:12]
            cpu_icu_sel = peri_access_en
            peri_mmu_data = icu_cpu_data
            peri_mmu_rdy = icu_cpu_rdy
            peri_mmu_berr = icu_cpu_berr
            all_other_selects = 0
        when CPR_base[17:12]
            cpu_cpr_sel = peri_access_en
            peri_mmu_data = cpr_cpu_data
            peri_mmu_rdy = cpr_cpu_rdy
            peri_mmu_berr = cpr_cpu_berr
            all_other_selects. = 0
        when ROM_base[17:12]
            cpu_rom_sel = peri_access_en
            peri_mmu_data = rom_cpu_data
            peri_mmu_rdy = rom_cpu_rdy
            peri_mmu_berr = rom_cpu_berr
            all_other_selects = 0
```

```
        when PSS_base[17:12]
            cpu_pss_sel = peri_access_en
            peri_mmu_data = pss_cpu_data
            peri_mmu_rdy = pss_cpu_rdy
            peri_mmu_berr = pss_cpu_berr
            all_other_selects = 0
        when DIU_base[17:12]
            cpu_diu_sel = peri_access_en
            peri_mmu_data = diu_cpu_data
            peri_mmu_rdy = diu_cpu_rdy
            peri_mmu_berr = diu_cpu_berr
            all_other_selects = 0
        when PCU_base[17:12]
            cpu_pcu_sel = peri_access_en
            peri_mmu_data = pcu_cpu_data
            peri_mmu_rdy = pcu_cpu_rdy
            peri_mmu_berr = pcu_cpu_berr
            all_other_selects = 0
        when others
            all_block_selects = 0
            peri_mmu_data = 0x00000000
            peri_mmu_rdy = 0
            peri_mmu_berr = 1
        end case
```

11.6.6.3 MMU Control Block

The MMU Control Block determines whether every CPU access is a valid access. No more than one cycle is to be consumed in determining the validity of an access and all accesses must terminate with the assertion of either *mmu_cpu_rdy* or *mmu_cpu_berr*. To safeguard against stalling the CPU a simple bus timeout mechanism will be supported.

Table 24. MMU Control Block I/Os

Port name	Pins	I/O	Description
Global SoPEC signals			
prst_n	1	In	Global reset. Synchronous to *pclk*, active low.
pclk	1	In	Global clock
Toplevel/Common MMU Control Block signals			
cpu_adr[21:2]	22	Out	Address bus for both DRAM and peripheral access.
cpu_acode[1:0]	2	Out	CPU access code signals (*cpu_mmu_acode*) retimed to meet the CPU Subsystem Bus timing requirements
dram_access_en	1	Out	DRAM Access Enable signal. Indicates that the current CPU access is a valid DRAM access.
MMU Control Block to LEON AHB bridge signals			

cpu_mmu_adr[31:0]	32	In	CPU core address bus.
cpu_dataout[31:0]	32	In	Toplevel CPU data bus
mmu_cpu_data[31:0]	32	Out	Data bus to the CPU core. Carries the data for all CPU read operations
cpu_rwn	1	In	Toplevel CPU Read/notWrite signal.
cpu_mmu_acode[1:0]	2	In	CPU access code signals
mmu_cpu_rdy	1	Out	Ready signal to the CPU core. Indicates the completion of all valid CPU accesses.
mmu_cpu_berr	1	Out	Bus Error signal to the CPU core. This signal is asserted to terminate an invalid access.
cpu_start_access	1	In	Start Access signal from the LEON AHB bridge indicating the start of a data transfer and that the *cpu_adr, cpu_dataout, cpu_rwn* and *cpu_acode* signals are all valid. This signal is only asserted during the first cycle of an access.
cpu_iack	1	In	Interrupt Acknowledge signal from the CPU. This signal is only asserted during an interrupt acknowledge cycle.
cpu_ben[1:0]	2	In	Byte enable signals indicating which bytes of the 32-bit bus are being accessed.
MMU Control Block to CPU Subsystem Bus Interface signals			
cpu_adr[17:12]	8	Out	Toplevel CPU Address bus. Only bits 17-12 are required to decode the peripherals address space
peri_access_en	1	Out	Enable Access signal. A peripheral access cannot be initiated unless it has been enabled by the MMU Control Unit
peri_mmu_data[31:0]	32	In	Data bus from the selected peripheral
peri_mmu_rdy	1	In	Data Ready signal. Indicates the data on the *peri_mmu_data* bus is valid for a read cycle or that the data was successfully written to the peripheral for a write cycle.
peri_mmu_berr	1	In	Bus Error signal. Indicates a bus error has occurred in accessing the selected peripheral

Description:

The MMU Control Block is responsible for the MMU's core functionality, namely determining whether or not an access to any part of the address map is valid. An access is considered valid if it is to a mapped area of the address space and if the CPU is running in the appropriate mode for that address space. Furthermore the MMU control block must correctly handle the special cases that are: an interrupt acknowledge cycle, a reset exception vector fetch, an access that crosses a 256-

bit DRAM word boundary and a bus timeout condition. The following pseudocode shows the logic required to implement the MMU Control Block functionality. It does not deal with the timing relationships of the various signals - it is the designer's responsibility to ensure that these relationships are correct and comply with the different bus protocols. For simplicity the pseudocode

5   is split up into numbered sections so that the functionality may be seen more easily.

It is important to note that the style used for the pseudocode will differ from the actual coding style used in the RTL implementation. The pseudocode is only intended to capture the required functionality, to clearly show the criteria that need to be tested rather than to describe how the implementation should be performed. In particular the different comparisons of the address used to

10   determine which part of the memory map, which DRAM region (if applicable) and the permission checking should all be performed in parallel (with results ORed together where appropriate) rather than sequentially as the pseudocode implies.

PS0 Description: This first segment of code defines a number of constants and variables that are used elsewhere in this description. Most signals have been defined in the I/O descriptions of the

15   MMU sub-blocks that precede this section of the document. The *post_reset_state* variable is used later (in section PS4) to determine if we should trap a null pointer access.

```
PS0:
         const UnusedBottom = 0x002AC000
         const DRAMTop = 0x4027FFFF
         const UserDataSpace = b01
         const UserProgramSpace = b00
         const SupervisorDataSpace = b11
         const SupervisorProgramSpace = b10
         const ResetExceptionCycles = 0x2

         cpu_adr_peri_masked[5:0] = cpu_mmu_adr[17:12]
         cpu_adr_dram_masked[16:0] = cpu_mmu_adr & 0x003FFFE0

         if (prst_n == 0) then      // Initialise everything
            cpu_adr = cpu_mmu_adr[21:2]
            peri_access_en = 0
            dram_access_en = 0
            mmu_cpu_data = peri_mmu_data
            mmu_cpu_rdy = 0
            mmu_cpu_berr = 0
            post_reset_state = TRUE
            access_initiated = FALSE
            cpu_access_cnt = 0

            // The following is used to determine if we are coming out
         of reset for the purposes of
            // reset exception vector redirection. There may be a
         convenient signal in the CPU core
```

```
            // that we could use instead of this.
            if  ((cpu_start_access   ==   1)   AND   (cpu_access_cnt   <
         ResetExceptionCycles) AND
                  (clock_tick == TRUE)) then
                cpu_access_cnt = cpu_access_cnt +1
            else
                post_reset_state = FALSE
```

PS1 Description: This section is at the top of the hierarchy that determines the validity of an access.
The address is tested to see which macro-region (i.e. Unused, CPU Subsystem or DRAM) it falls
into or whether the reset exception vector is being accessed.

```
         PS1:
            if (cpu_mmu_adr >= UnusedBottom) then
                  // The access is to an invalid area of the address
            space. See section PS2

            elsif   ((cpu_mmu_adr   >   DRAMTop)   AND   (cpu_mmu_adr   <
            UnusedBottom)) then
                  // We are in the CPU Subsystem/PEP Subsystem address
            space. See section PS3

            // Only remaining possibility is an access to DRAM address
            space
            // First we need to intercept the special case for the
            reset exception vector

            elsif (cpu_mmu_adr < 0x00000010) then
                  // The reset exception is being accessed. See section PS4

            elsif    ((cpu_adr_dram_masked    >=    Region0Bottom)    AND
            (cpu_adr_dram_masked <=
                     Region0Top) ) then
                  // We are in Region0. See section PS5

            elsif    ((cpu_adr_dram_masked    >=    RegionNBottom)    AND
            (cpu_adr_dram_masked <=
                     RegionNTop) ) then   // we are in RegionN
                     // Repeat the Region0 (i.e. section PS5) logic for
            each of Region1 to Region7

            else    // We could end up here if there were gaps in the
            DRAM regions
                peri_access_en = 0
                dram_access_en = 0
```

```
                    mmu_cpu_berr = 1      // we have an unknown access error,
             most likely due to hitting
                    mmu_cpu_rdy = 0       // a gap in the DRAM regions


5                   // Only thing remaining is to implement a bus timeout
             function. This is done in PS6


             end
```

PS2 Description: Accesses to the large unused area of the address space are trapped by this
section. No bus transactions are initiated and the *mmu_cpu_berr* signal is asserted.

```
             PS2:
                 elsif (cpu_mmu_adr >= UnusedBottom) then
                     peri_access_en = 0  // The access is to an invalid area
                 of the address space
                     dram_access_en = 0
                     mmu_cpu_berr = 1
                     mmu_cpu_rdy = 0
```

PS3 Description: This section deals with accesses to CPU Subsystem peripherals, including the
MMU itself. If the MMU registers are being accessed then no external bus transactions are required.
Access to the MMU registers is only permitted if the CPU is making a data access from supervisor
mode, otherwise a bus error is asserted and the access terminated. For non-MMU accesses then
transactions occur over the CPU Subsystem Bus and each peripheral is responsible for determining
whether or not the CPU is in the correct mode (based on the *cpu_acode* signals) to be permitted
access to its registers. Note that all of the PEP registers are accessed via the PCU which is on the
CPU Subsystem Bus.

```
             PS3:
                 elsif ((cpu_mmu_adr > DRAMTop) AND (cpu_mmu_adr <
                 UnusedBottom)) then
                     // We are in the CPU Subsystem/PEP Subsystem address
                 space

                     cpu_adr = cpu_mmu_adr[21:2]
                     if (cpu_adr_peri_masked == MMU_base) then  // access is
                 to local registers
                         peri_access_en = 0
                         dram_access_en = 0
                         if (cpu_acode == SupervisorDataSpace) then
                             for (i=0; i<26; i++) {
                                 if ((i == cpu_mmu_adr[6:2]) then  // selects the
                 addressed register
                                     if (cpu_rwn == 1) then
```

```
                        mmu_cpu_data[16:0] = MMUReg[i]     // MMUReg[i]
            is one of the
                        mmu_cpu_rdy = 1                    // registers
            in Table
                        mmu_cpu_berr = 0
                    else   // write cycle
                    MMUReg[i] = cpu_dataout[16:0]
                    mmu_cpu_rdy = 1
                    mmu_cpu_berr = 0
                else    // there  is  no  register  mapped  to  this
            address
                        mmu_cpu_berr = 1    // do  we  really  want  a
            bus_error here as registers
                        mmu_cpu_rdy = 0   // are  just  mirrored  in  other
            blocks


            else // we have an access violation
                mmu_cpu_berr = 1
                mmu_cpu_rdy = 0


        else // access is to something else on the CPU Subsystem
    Bus
            peri_access_en = 1
            dram_access_en = 0
            mmu_cpu_data = peri_mmu_data
            mmu_cpu_rdy = peri_mmu_rdy
            mmu_cpu_berr = peri_mmu_berr
```

PS4 Description: The only correct accesses to the locations beneath 0x00000010 are fetches of the reset trap handling routine and these should be the first accesses after reset: Here we trap all other accesses to these locations regardless of the CPU mode. The most likely cause of such an access will be the use of a null pointer in the program executing on the CPU.

PS4:
```
        elsif (cpu_mmu_adr < 0x00000010) then
            if (post_reset_state == TRUE)) then
                cpu adr = cpu mmu adr[21:2]
                peri_access_en = 1
                dram_access_en = 0
                mmu_cpu_data = peri_mmu_data
                mmu_cpu_rdy = peri_mmu_rdy
                mmu_cpu_berr = peri_mmu_berr
            else    // we  have  a  problem  (almost  certainly  a  null
            pointer)
                    peri_access_en = 0
```

```
                              dram_access_en = 0
                              mmu_cpu_berr = 1
                              mmu_cpu_rdy = 0
```

5   PS5 Description: This large section of pseudocode simply checks whether the access is within the
bounds of DRAM Region0 and if so whether or not the access is of a type permitted by the
Region0Control register. If the access is permitted then a DRAM access is initiated. If the access is
not of a type permitted by the *Region0Control* register then the access is terminated with a bus
error.

10

PS5:
```
      elsif    ((cpu_adr_dram_masked    >=    Region0Bottom)    AND
(cpu_adr_dram_masked <=
              Region0Top) ) then   // we are in Region0
```

15
```
      cpu_adr = cpu_mmu_adr[21:2]
      if (cpu_rwn == 1) then
          if    ((cpu_acode    ==    SupervisorProgramSpace    AND
Region0Control[2] == 1))
```

20
```
              OR    (cpu_acode    ==    UserProgramSpace    AND
Region0Control[5] == 1)) then
                              // this  is  a  valid  instruction
fetch from Region0
                              // The  dram_cpu_data  bus  goes
```

25
```
directly to the LEON
                              // AHB  bridge  which  also  handles
the hready generation
              peri_access_en = 0
              dram_access_en = 1
```

30
```
              mmu_cpu_berr = 0

          elsif    ((cpu_acode    ==    SupervisorDataSpace    AND
Region0Control[0] == 1)
              OR    (cpu_acode    ==    UserDataSpace    AND
```

35
```
Region0Control[3] == 1)) then
                              // this  is  a  valid
read access from Region0
              peri_access_en = 0
              dram_access_en = 1
```

40
```
              mmu_cpu_berr = 0

          else                              // we have an access
violation
              peri_access_en = 0
```

45
```
              dram_access_en = 0
```

```
                                    mmu_cpu_berr = 1
                                    mmu_cpu_rdy = 0


                        else                          // it is a write access
5                           if      ((cpu_acode      ==      SupervisorDataSpace      AND
                        Region0Control[1] == 1)
                                OR       (cpu_acode      ==      UserDataSpace      AND
                        Region0Control[4] == 1)) then
                                                        //  this  is  a  valid
10                      write access to Region0
                                    peri_access_en = 0
                                    dram_access_en = 1
                                    mmu_cpu_berr = 0
                                else                          // we  have  an  access
15                      violation
                                    peri_access_en = 0
                                    dram_access_en = 0
                                    mmu_cpu_berr = 1
                                    mmu_cpu_rdy = 0

20
```

PS6 Description: This final section of pseudocode deals with the special case of a bus timeout. This occurs when an access has been initiated but has not completed before the *BusTimeout* number of *pclk* cycles. While access to both DRAM and CPU/PEP Subsystem registers will take a variable number of cycles (due to DRAM traffic, PCU command execution or the different timing required to access registers in imported IP) each access should complete before a timeout occurs. Therefore it should not be possible to stall the CPU by locking either the CPU Subsystem or DIU buses. However given the fatal effect such a stall would have it is considered prudent to implement bus timeout detection.

```
30              PS6:
                    // Only  thing  remaining  is  to  implement  a  bus  timeout
                    function.

                    if ((cpu_start_access == 1) then
35                      access_initiated = TRUE
                        timeout_countdown = BusTimeout

                    if ((mmu_cpu_rdy == 1 ) OR (mmu_cpu_berr ==1 )) then
                        access_initiated = FALSE
40                      peri_access_en = 0
                        dram_access_en = 0

                    if ((clock_tick == TRUE) AND (access_initiated == TRUE) AND
                    (BusTimeout != 0))
45                      if (timeout_countdown > 0) then
```

```
                           timeout_countdown--
                       else // timeout has occurred
                           peri_access_en = 0          // abort the access
                           dram_access_en = 0
                           mmu_cpu_berr = 1
                           mmu_cpu_rdy = 0
```

11.7 LEON CACHES

The version of LEON implemented on SoPEC features 1 kB of ICache and 1 kB of DCache. Both caches are direct mapped and feature 8 word lines so their data RAMs are arranged as 32 x 256-bit and their tag RAMs as 32 x 30-bit (itag) or 32 x 32-bit (dtag). Like most of the rest of the LEON code used on SoPEC the cache controllers are taken from the leon2-1.0.7 release. The LEON cache controllers and cache RAMs have been modified to ensure that an entire 256-bit line is refilled at a time to make maximum use out of the memory bandwidth offered by the embedded DRAM organization (DRAM lines are also 256-bit). The data cache controller has also been modified to ensure that user mode code cannot access the DCache contents unless it is authorised to do so. A block diagram of the LEON CPU core as implemented on SoPEC is shown in Figure 23 below.

In this diagram dotted lines are used to indicate hierarchy and red items represent signals or wrappers added as part of the SoPEC modifications. LEON makes heavy use of VHDL records and the records used in the CPU core are described in Table 25. Unless otherwise stated the records are defined in the iface.vhd file (part of the LEON release) and this should be consulted for a complete breakdown of the record elements.

Table 25. Relevant LEON records

Record Name	Description
rfi	Register File Input record. Contains address, datain and control signals for the register file.
rfo	Register File Output record. Contains the data out of the dual read port register file.
ici	Instruction Cache In record. Contains program counters from different stages of the pipeline and various control signals
ico	Instruction Cache Out record. Contains the fetched instruction data and various control signals. This record is also sent to the DCache (i.e. icol) so that diagnostic accesses (e.g. *lda/sta*) can be serviced.
dci	Data Cache In record. Contains address and data buses from different stages of the pipeline (execute & memory) and various control signals
dco	Data Cache Out record. Contains the data retrieved from either memory or the caches and various control signals. This record is also sent to the ICache (i.e. dcol) so that diagnostic accesses (e.g. *lda/sta*) can be serviced.
iui	Integer Unit In record. This record contains the interrupt request level and a record for use with LEONs Debug Support Unit (DSU)

96

iuo	Integer Unit Out record. This record contains the acknowledged interrupt request level with control signals and a record for use with LEONs Debug Support Unit (DSU)
mcii	Memory to Cache Icache In record. Contains the address of an Icache miss and various control signals
mcio	Memory to Cache Icache Out record. Contains the returned data from memory and various control signals
mcdi	Memory to Cache Dcache In record. Contains the address and data of a Dcache miss or write and various control signals
mcdo	Memory to Cache Dcache Out record. Contains the returned data from memory and various control signals
ahbi	AHB In record. This is the input record for an AHB master and contains the data bus and AHB control signals. The destination for the signals in this record is the AHB controller. This record is defined in the amba.vhd file
ahbo	AHB Out record. This is the output record for an AHB master and contains the address and data buses and AHB control signals. The AHB controller drives the signals in this record. This record is defined in the amba.vhd file
ahbsi	AHB Slave In record. This is the input record for an AHB slave and contains the address and data buses and AHB control signals. It is used by the DCache to facilitate cache snooping (this feature is not enabled in SoPEC). This record is defined in the amba.vhd file
crami	Cache RAM In record. This record is composed of records of records which contain the address, data and tag entries with associated control signals for both the ICache RAM and DCache RAM
cramo	Cache RAM Out record. This record is composed of records of records which contain the data and tag entries with associated control signals for both the ICache RAM and DCache RAM
iline_rdy	Control signal from the ICache controller to the instruction cache memory. This signal is active (high) when a full 256-bit line (on *dram_cpu_data*) is to be written to cache memory.
dline_rdy	Control signal from the DCache controller to the data cache memory. This signal is active (high) when a full 256-bit line (on dram_cpu_data) is to be written to cache memory.
dram_cpu_data	256-bit data bus from the embedded DRAM

11.7.1   Cache controllers

The LEON cache module consists of three components: the ICache controller (icache.vhd), the DCache controller (dcache.vhd) and the AHB bridge (acache.vhd) which translates all cache misses into memory requests on the AHB bus.

5   In order to enable full line refill operation a few changes had to be made to the cache controllers. The ICache controller was modified to ensure that whenever a location in the cache was updated

(i.e. the cache was enabled and was being refilled from DRAM) all locations on that cache line had their valid bits set to reflect the fact that the full line was updated. The *iline_rdy* signal is asserted by the ICache controller when this happens and this informs the cache wrappers to update all locations in the idata RAM for that line.

5   A similar change was made to the DCache controller except that the entire line was only updated following a read miss and that existing write through operation was preserved. The DCache controller uses the *dline_rdy* signal to instruct the cache wrapper to update all locations in the ddata RAM for a line. An additional modification was also made to ensure that a double-word load instruction from a non-cached location would only result in one read access to the DIU i.e. the

10   second read would be serviced by the data cache. Note that if the DCache is turned off then a double-word load instruction will cause two DIU read accesses to occur even though they will both be to the same 256-bit DRAM line.

The DCache controller was further modified to ensure that user mode code cannot access cached data to which it does not have permission (as determined by the relevant *RegionNControl* register

15   settings at the time the cache line was loaded). This required an extra 2 bits of tag information to record the user read and write permissions for each cache line. These user access permissions can be updated in the same manner as the other tag fields (i.e. address and valid bits) namely by line refill, STA instruction or cache flush. The user access permission bits are checked every time user code attempts to access the data cache and if the permissions of the access do not agree with the

20   permissions returned from the tag RAM then a cache miss occurs. As the MMU evaluates the access permissions for every cache miss it will generate the appropriate exception for the forced cache miss caused by the errant user code. In the case of a prohibited read access the trap will be immediate while a prohibited write access will result in a deferred trap. The deferred trap results from the fact that the prohibited write is committed to a write buffer in the DCache controller and

25   program execution continues until the prohibited write is detected by the MMU which may be several cycles later. Because the errant write was treated as a write miss by the DCache controller (as it did not match the stored user access permissions) the cache contents were not updated and so remain coherent with the DRAM contents (which do not get updated because the MMU intercepted the prohibited write). Supervisor mode code is not subject to such checks and so has

30   free access to the contents of the data cache.

In addition to AHB bridging, the ACache component also performs arbitration between ICache and DCache misses when simultaneous misses occur (the DCache always wins) and implements the Cache Control Register (CCR). The leon2-1.0.7 release is inconsistent in how it handles cacheability: For instruction fetches the cacheability (i.e. is the access to an area of memory that is

35   cacheable) is determined by the ICache controller while the ACache determines whether or not a data access is cacheable. To further complicate matters the DCache controller does determine if an access resulting from a cache snoop by another AHB master is cacheable (Note that the SoPEC ASIC does not implement cache snooping as it has no need to do so). This inconsistency has been cleaned up in more recent LEON releases but is preserved here to minimise the number of changes

to the LEON RTL. The cache controllers were modified to ensure that only DRAM accesses (as defined by the SoPEC memory map) are cached.

The only functionality removed as a result of the modifications was support for burst fills of the ICache. When enabled burst fills would refill an ICache line from the location where a miss occurred up to the end of the line. As the entire line is now refilled at once (when executing from DRAM) this functionality is no longer required. Furthermore more substantial modifications to the ICache controller would be needed if we wished to preserve this function without adversely affecting full line refills. The CCR was therefore modified to ensure that the instruction burst fetch bit (bit16) was tied low and could not be written to.

11.7.1.1 LEON Cache Control Register

The CCR controls the operation of both the I and D caches. Note that the bitfields used on the SoPEC implementation of this register are based on the LEON v1.0.7 implementation and some bits have their values tied off. See section 4 of the LEON manual for a description of the LEON cache controllers.

Table 26. LEON Cache Control Register

Field Name	bit(s)	Description
ICS	1:0	Instruction cache state: 00 - disabled 01 - frozen 10 - disabled 11 - enabled
Reserved	13:6	Reserved. Reads as 0.
DCS	3:2	Data cache state: 00 - disabled 01 - frozen 10 - disabled 11 - enabled
IF	4	ICache freeze on interrupt 0 - Do not freeze the ICache contents on taking an interrupt 1 - Freeze the ICache contents on taking an interrupt
DF	5	DCache freeze on interrupt 0 - Do not freeze the DCache contents on taking an interrupt 1 - Freeze the DCache contents on taking an interrupt
Reserved	13:6	Reserved. Reads as 0.
DP	14	Data cache flush pending. 0 - No DCache flush in progress 1 - DCache flush in progress This bit is ReadOnly.

IP	15	Instruction cache flush pending. 0 - No ICache flush in progress 1 - ICache flush in progress This bit is ReadOnly.
IB	16	Instruction burst fetch enable. This bit is tied low on SoPEC because it would interfere with the operation of the cache wrappers. Burst refill functionality is automatically provided in SoPEC by the cache wrappers.
Reserved	20:17	Reserved. Reads as 0.
FI	21	Flush instruction cache. Writing a 1 this bit will flush the ICache. Reads as 0.
FD	22	Flush data cache. Writing a 1 this bit will flush the DCache. Reads as 0.
DS	23	Data cache snoop enable. This bit is tied low in SoPEC as there is no requirement to snoop the data cache.
Reserved	31:24	Reserved. Reads as 0.

11.7.2   Cache wrappers

The cache RAMs used in the leon2-1.0.7 release needed to be modified to support full line refills and the correct IBM macros also needed to be instantiated. Although they are described as RAMs throughout this document (for consistency), register arrays are actually used to implement the cache RAMs. This is because IBM SRAMs were not available in suitable configurations (offered configurations were too big) to implement either the tag or data cache RAMs. Both instruction and data tag RAMs are implemented using dual port (1 Read & 1 Write) register arrays and the clocked write-through versions of the register arrays were used as they most closely approximate the single port SRAM LEON expects to see.

11.7.2.1 *Cache Tag RAM wrappers*

The itag and dtag RAMs differ only in their width - the itag is a 32x30 array while the dtag is a 32x32 array with the extra 2 bits being used to record the user access permissions for each line. When read using a LDA instruction both tags return 32-bit words. The tag fields are described in Table 27 and Table 28 below. Using the IBM naming conventions the register arrays used for the tag RAMs are called RA032X30D2P2W1R1M3 for the itag and RA032X32D2P2W1R1M3 for the dtag. The *ibm_syncram* wrapper used for the tag RAMs is a simple affair that just maps the wrapper ports on to the appropriate ports of the IBM register array and ensures the output data has the correct timing by registering it. The tag RAMs do not require any special modifications to handle full line refills.

Table 27. LEON Instruction Cache Tag

Field Name	bit(s)	Description
Valid	7:0	Each valid bit indicates whether or not the corresponding word of the cache line contains valid data

		Reserved - these bits do not exist in the itag RAM. Reads as
Reserved	9:8	0.
Address	31:10	The tag address of the cache line

Table 28. LEON Data Cache Tag

Field Name	bit(s)	Description
Valid	7:0	Each valid bit indicates whether or not the corresponding word of the cache line contains valid data
URP	8	User read permission. 0 - User mode reads will force a refill of this line 1 - User mode code can read from this cache line.
UWP	9	User write permission. 0 - User mode writes will not be written to the cache 1 - User mode code can write to this cache line.
Address	31:10	The tag address of the cache line

11.7.2.2 Cache Data RAM wrappers

5   The cache data RAM contains the actual cached data and nothing else. Both the instruction and data cache data RAMs are implemented using 8 32x32-bit register arrays and some additional logic to support full line refills. Using the IBM naming conventions the register arrays used for the tag RAMs are called RA032X32D2P2W1R1M3. The *ibm_cdram_wrap* wrapper used for the tag RAMs is shown in Figure 24 below.

10   To the cache controllers the cache data RAM wrapper looks like a 256x32 single port SRAM (which is what they expect to see) with an input to indicate when a full line refill is taking place (the *line_rdy* signal). Internally the 8-bit address bus is split into a 5-bit lineaddress, which selects one of the 32 256-bit cache lines, and a 3-bit wordaddress which selects one of the 8 32-bit words on the cache line. Thus each of the 8 32x32 register arrays contains one 32-bit word of each cache line. When a

15   full line is being refilled (indicated by both the *line_rdy* and *write* signals being high) every register array is written to with the appropriate 32 bits from the *linedatain* bus which contains the 256-bit line returned by the DIU after a cache miss. When just one word of the cache line is to be written (indicated by the *write* signal being high while the *line_rdy* is low) then the wordaddress is used to enable the write signal to the selected register array only - all other write enable signals are kept

20   low. The data cache controller handles byte and half-word write by means of a read-modify-write operation so writes to the cache data RAM are always 32-bit.

The wordaddress is also used to select the correct 32-bit word from the cache line to return to the LEON integer unit.

11.8   REALTIME DEBUG UNIT (RDU)

25   The RDU facilitates the observation of the contents of most of the CPU addressable registers in the SoPEC device in addition to some pseudo-registers in realtime. The contents of pseudo-registers, i.e. registers that are collections of otherwise unobservable signals and that do not affect the

101

functionality of a circuit, are defined in each block as required. Many blocks do not have pseudo-registers and some blocks (e.g. ROM, PSS) do not make debug information available to the RDU as it would be of little value in realtime debug.

Each block that supports realtime debug observation features a *DebugSelect* register that controls a local mux to determine which register is output on the block's data bus (i.e. *block_cpu_data*). One small drawback with reusing the blocks data bus is that the debug data cannot be present on the same bus during a CPU read from the block. An accompanying active high *block_cpu_debug_valid* signal is used to indicate when the data bus contains valid debug data and when the bus is being used by the CPU. There is no arbitration for the bus as the CPU will always have access when required. A block diagram of the RDU is shown in Figure 25.

Table 29. RDU I/Os

Port name	Pins	I/O	Description
diu_cpu_data	32	In	Read data bus from the DIU block
cpr_cpu_data	32	In	Read data bus from the CPR block
gpio_cpu_data	32	In	Read data bus from the GPIO block
icu_cpu_data	32	In	Read data bus from the ICU block
lss_cpu_data	32	In	Read data bus from the LSS block
pcu_cpu_debug_data	32	In	Read data bus from the PCU block
scb_cpu_data	32	In	Read data bus from the SCB block
tim_cpu_data	32	In	Read data bus from the TIM block
diu_cpu_debug_valid	1	In	Signal indicating the data on the *diu_cpu_data* bus is valid debug data.
tim_cpu_debug_valid	1	In	Signal indicating the data on the *tim_cpu_data* bus is valid debug data.
scb_cpu_debug_valid	1	In	Signal indicating the data on the *scb_cpu_data* bus is valid debug data.
pcu_cpu_debug_valid	1	In	Signal indicating the data on the *pcu_cpu_data* bus is valid debug data.
lss_cpu_debug_valid	1	In	Signal indicating the data on the *lss_cpu_data* bus is valid debug data.
icu_cpu_debug_valid	1	In	Signal indicating the data on the *icu_cpu_data* bus is valid debug data.
gpio_cpu_debug_valid	1	In	Signal indicating the data on the *gpio_cpu_data* bus is valid debug data.
cpr_cpu_debug_valid	1	In	Signal indicating the data on the *cpr_cpu_data* bus is valid debug data.
debug_data_out	32	Out	Output debug data to be muxed on to the PHI/GPIO/other pins

debug_data_valid	1	Out	Debug valid signal indicating the validity of the data on *debug_data_out*. This signal is used in all debug configurations
debug_cntrl	33	Out	Control signal for each debug data line indicating whether or not the debug data should be selected by the pin mux

As there are no spare pins that can be used to output the debug data to an external capture device some of the existing I/Os will have a debug multiplexer placed in front of them to allow them be used as debug pins. Furthermore not every pin that has a debug mux will always be available to

5 carry the debug data as they may be engaged in their primary purpose e.g. as a GPIO pin. The RDU therefore outputs a *debug_cntrl* signal with each debug data bit to indicate whether the mux associated with each debug pin should select the debug data or the normal data for the pin. The *DebugPinSel1* and *DebugPinSel2* registers are used to determine which of the 33 potential debug pins are enabled for debug at any particular time.

10 As it may not always be possible to output a full 32-bit debug word every cycle the RDU supports the outputting of an n-bit sub-word every cycle to the enabled debug pins. Each debug test would then need to be re-run a number of times with a different portion of the debug word being output on the n-bit sub-word each time. The data from each run should then be correlated to create a full 32-bit (or whatever size is needed) debug word for every cycle. The *debug_data_valid* and *pclk_out*

15 signals will accompany every sub-word to allow the data to be sampled correctly. The *pclk_out* signal is sourced close to its output pad rather than in the RDU to minimise the skew between the rising edge of the debug data signals (which should be registered close to their output pads) and the rising edge of *pclk_out*.

As multiple debug runs will be needed to obtain a complete set of debug data the n-bit sub-word will

20 need to contain a different bit pattern for each run. For maximum flexibility each debug pin has an associated *DebugDataSrc* register that allows any of the 32 bits of the debug data word to be output on that particular debug data pin. The debug data pin must be enabled for debug operation by having its corresponding bit in the *DebugPinSel* registers set for the selected debug data bit to appear on the pin.

25 The size of the sub-word is determined by the number of enabled debug pins which is controlled by the *DebugPinSel* registers. Note that the *debug_data_valid* signal is always output. Furthermore *debug_cntrl[0]* (which is configured by *DebugPinSel1*) controls the mux for both the *debug_data_valid* and *pclk_out* signals as both of these must be enabled for any debug operation. The mapping of *debug_data_out[n]* signals onto individual pins will take place outside the RDU.

30 This mapping is described in Table 30 below.

Table 30. DebugPinSel mapping

bit #	Pin
DebugPinSel1	phi_frclk. The *debug_data_valid* signal will appear

				on this pin when enabled. Enabling this pin also automatically enables the phi_readl pin which will output the *pclk_out* signal
DebugPinSel2(0-31)	gpio[0...31]			

Table 31. RDU Configuration Registers

Address offset from MMU_base	Register	#bits	Reset	Description
0x80	DebugSrc	4	0x00	Denotes which block is supplying the debug data. The encoding of this block is given below. 0 - MMU 1 - TIM 2 - LSS 3- GPIO 4 - SCB 5 - ICU 6 - CPR 7 - DIU 8 – PCU
0x84	DebugPinSel 1	1	0x0	Determines whether the phi_frclk and phi_readl pins are used for debug output. 1 - Pin outputs debug data 0 - Normal pin function
0x88	DebugPinSel 2	32	0x000 0_000 0	Determines whether a pin is used for debug data output. 1 - Pin outputs debug data 0 - Normal pin function
0x8C to 0x108	DebugDataSr c[31:0]	32 x 5	0x00	Selects which bit of the 32-bit debug data word will be output on debug_data_out[N]

11.9   INTERRUPT OPERATION

The interrupt controller unit (see chapter 14) generates an interrupt request by driving interrupt request lines with the appropriate interrupt level. LEON supports 15 levels of interrupt with level 15 as the highest level (the SPARC architecture manual [36] states that level 15 is non-maskable but we have the freedom to mask this if desired). The CPU will begin processing an interrupt exception when execution of the current instruction has completed and it will only do so if the interrupt level is higher than the current processor priority. If a second interrupt request arrives with the same level as an executing interrupt service routine then the exception will not be processed until the executing routine has completed.

When an interrupt trap occurs the LEON hardware will place the program counters (PC and nPC) into two local registers. The interrupt handler routine is expected, as a minimum, to place the PSR register in another local register to ensure that the LEON can correctly return to its pre-interrupt state. The 4-bit interrupt level (*irl*) is also written to the trap type (*tt*) field of the TBR (Trap Base Register) by hardware. The TBR then contains the vector of the trap handler routine the processor will then jump. The TBA (Trap Base Address) field of the TBR must have a valid value before any interrupt processing can occur so it should be configured at an early stage.

Interrupt pre-emption is supported while ET (Enable Traps) bit of the PSR is set. This bit is cleared during the initial trap processing. In initial simulations the ET bit was observed to be cleared for up to 30 cycles. This causes significant additional interrupt latency in the worst case where a higher priority interrupt arrives just as a lower priority one is taken.

The interrupt acknowledge cycles shown in Figure 26 below are derived from simulations of the LEON processor. The SoPEC toplevel interrupt signals used in this diagram map directly to the LEON interrupt signals in the *iui* and *iuo* records. An interrupt is asserted by driving its (encoded) level on the *icu_cpu_ilevel[3:0]* signals (which map to *iui.irl[3:0]*). The LEON core responds to this, with variable timing, by reflecting the level of the taken interrupt on the *cpu_icu_ilevel[3:0]* signals (mapped to *iuo.irl[3:0]*) and asserting the acknowledge signal *cpu_iack (iuo.intack)*.The interrupt controller then removes the interrupt level one cycle after it has seen the level been acknowledged by the core. If there is another pending interrupt (of lower priority) then this should be driven on *icu_cpu_ilevel[3:0]* and the CPU will take that interrupt (the level 9 interrupt in the example below) once it has finished processing the higher priority interrupt. The *cpu_icu_ilevel[3:0]* signals always reflect the level of the last taken interrupt, even when the CPU has finished processing all interrupts.

11.10   BOOT OPERATION

See section 17.2 for a description of the SoPEC boot operation.

11.11   SOFTWARE DEBUG

Software debug mechanisms are discussed in the "SoPEC Software Debug" document [15].

12 Serial Communications Block (SCB)

12.1   OVERVIEW

The Serial Communications Block (SCB) handles the movement of all data between the SoPEC and the host device (e.g. PC) and between master and slave SoPEC devices. The main components of the SCB are a Full-Speed (FS) USB Device Core, a FS USB Host Core, a Inter-SoPEC Interface (ISI), a DMA manager, the SCB Map and associated control logic. The need for these components and the various types of communication they provide is evident in a multi-SoPEC printer configuration.

12.1.1   Multi-SoPEC systems

While single SoPEC systems are expected to form the majority of SoPEC systems the SoPEC device must also support its use in multi-SoPEC systems such as that shown in Figure 27. A SoPEC may be assigned any one of a number of identities in a multi-SoPEC system. A SoPEC may be one or more of a PrintMaster, a LineSyncMaster, an ISIMaster, a StorageSoPEC or an ISISlave SoPEC.

12.1.1.1 ISIMaster device

The ISIMaster is the only device that controls the common ISI lines (see Figure 30) and typically interfaces directly with the host. In most systems the ISIMaster will simply be the SoPEC connected to the USB bus. Future systems, however, may employ an ISI-Bridge chip to interface between the host and the ISI bus and in such systems the ISI-Bridge chip will be the ISIMaster. There can only be one ISIMaster on an ISI bus.

Systems with multiple SoPECs may have more than one host connection, for example there could be two SoPECs communicating with the external host over their FS USB links (this would of course require two USB cables to be connected), but still only one ISIMaster.

While it is not expected to be required, it is possible for a device to hand over its role as the ISIMaster to another device on the ISI i.e. the ISIMaster is not necessarily fixed.

12.1.1.2 PrintMaster device

The PrintMaster device is responsible for co-ordinating all aspects of the print operation. This includes starting the print operation in all printing SoPECs and communicating status back to the external host. When the ISIMaster is a SoPEC device it is also likely to be the PrintMaster as well. There may only be one PrintMaster in a system and it is most likely to be a SoPEC device.

12.1.1.3 LineSyncMaster device

The LineSyncMaster device generates the *lsync* pulse that all SoPECs in the system must synchronize their line outputs with. Any SoPEC in the system could act as a LineSyncMaster although the PrintMaster is probably the most likely candidate. It is possible that the LineSyncMaster may not be a SoPEC device at all - it could, for example, come from some OEM motor control circuitry. There may only be one LineSyncMaster in a system.

12.1.1.4 Storage device

For certain printer types it may be realistic to use one SoPEC as a storage device without using its print engine capability - that is to effectively use it as an ISI-attached DRAM. A storage SoPEC would receive data from the ISIMaster (most likely to be an ISI-Bridge chip) and then distribute it to the other SoPECs as required. No other type of data flow (e.g. ISISlave -> storage SoPEC -> ISISlave) would need to be supported in such a scenario. The SCB supports this functionality at no additional cost because the CPU handles the task of transferring outbound data from the embedded DRAM to the ISI transmit buffer. The CPU in a storage SoPEC will have almost nothing else to do.

12.1.1.5 ISISlave device

Multi-SoPEC systems will contain one or more ISISlave SoPECs. An ISISlave SoPEC is primarily used to generate dot data for the printhead IC it is driving. An ISISlave will not transmit messages on the ISI without first receiving permission to do so, via a ping packet (see section 12.4.4.6), from the ISIMaster

12.1.1.6 ISI-Bridge device

SoPEC is targeted at the low-cost small office / home office (SoHo) market. It may also be used in future systems that target different market segments which are likely to have a high speed interface capability. A future device, known as an ISI-Bridge chip, is envisaged which will feature both a high speed interface (such as High-Speed (HS) USB, Ethernet or IEEE1394) and one or more ISI

interfaces. The use of multiple ISI buses would allow the construction of independent print systems within the one printer. The ISI-Bridge would be the ISIMaster for each of the ISI buses it interfaces to.

12.1.1.7 External host

The external host is most likely (but is not required) to be, a PC. Any system that can act as a USB host or that can interface to an ISI-Bridge chip could be the external host. In particular, with the development of USB On-The-Go (USB OTG), it is possible that a number of USB OTG enabled products such as PDAs or digital cameras will be able to directly interface with a SoPEC printer.

12.1.1.8 External USB device

The external USB device is most likely (but is not required) to be, a digital camera. Any system that can act as a USB device could be connected as an external USB device. This is to facilitate printing in the absence of a PC.

12.1.2 Types of communication

12.1.2.1 Communications with external host

The external host communicates directly with the ISIMaster in order to print pages. When the ISIMaster is a SoPEC, the communications channel is FS USB.

12.1.2.1.1 External host to ISIMaster communication

The external host will need to communicate the following information to the ISIMaster device:

- Communications channel configuration and maintenance information
- Most data destined for PrintMaster, ISISlave or storage SoPEC devices. This data is simply relayed by the ISIMaster
- Mapping of virtual communications channels, such as USB endpoints, to ISI destination

12.1.2.1.2 ISIMaster to external host communication

The ISIMaster will need to communicate the following information to the external host:

- Communications channel configuration and maintenance information
- All data originating from the PrintMaster, ISISlave or storage SoPEC devices and destined for the external host. This data is simply relayed by the ISIMaster

12.1.2.1.3 External host to PrintMaster communication

The external host will need to communicate the following information to the PrintMaster device:

- Program code for the PrintMaster
- Compressed page data for the PrintMaster
- Control messages to the PrintMaster
- Tables and static data required for printing e.g. dead nozzle tables, dither matrices etc.
- Authenticatable messages to upgrade the printer's capabilities

12.1.2.1.4 PrintMaster to external host communication

The PrintMaster will need to communicate the following information to the external host:

- Printer status information (i.e. authentication results, paper empty/jammed etc.)
- Dead nozzle information
- Memory buffer status information
- Power management status

- Encrypted SoPEC_id for use in the generation of PRINTER_QA keys during factory programming

12.1.2.1.5   External host to ISISlave communication

All communication between the external host and ISISlave SoPEC devices must be direct (via a dedicated connection between the external host and the ISISlave) or must take place via the ISIMaster. In the case of a SoPEC ISIMaster it is possible to configure each individual USB endpoint to act as a control channel to an ISISlave SoPEC if desired, although the endpoints will be more usually used to transport data. The external host will need to communicate the following information to ISISlave devices over the comms/ISI:

- Program code for ISISlave SoPEC devices
- Compressed page data for ISISlave SoPEC devices
- Control messages to the ISISlave SoPEC (where a control channel is supported)
- Tables and static data required for printing e.g. dead nozzle tables, dither matrices etc.
- Authenticatable messages to upgrade the printer's capabilities

12.1.2.1.6   ISISlave to external host communication

All communication between the ISISlave SoPEC devices and the external host must take place via the ISIMaster. The ISISlave will need to communicate the following information to the external host over the comms/ISI:

- Responses to the external host's control messages (where a control channel is supported)
- Dead nozzle information from the ISISlave SoPEC.
- Encrypted SoPEC_id for use in the generation of PRINTER_QA keys during factory programming

12.1.2.2 Communication with external USB device

12.1.2.2.1   ISIMaster to External USB device communication

- Communications channel configuration and maintenance information.

12.1.2.2.2   External USB device to ISIMaster communication

- Print data from a function on the external USB device.

12.1.2.3 Communication over ISI

12.1.2.3.1   ISIMaster to PrintMaster communication

The ISIMaster and PrintMaster will often be the same physical device. When they are different devices then the following information needs to be exchanged over the ISI:

- All data from the external host destined for the PrintMaster (see section 12.1.2.1.4). This data is simply relayed by the ISIMaster

12.1.2.3.2   PrintMaster to ISIMaster communication

The ISIMaster and PrintMaster will often be the same physical device. When they are different devices then the following information needs to be exchanged over the ISI:

- All data from the PrintMaster destined for the external host (see section 12.1.2.1.4). This data is simply relayed by the ISIMaster

12.1.2.3.3   ISIMaster to ISISlave communication

The ISIMaster may wish to communicate the following information to the ISISlaves:

- All data (including program code such as ISIId enumeration) originating from the external host and destined for the ISISlave (see section 12.1.2.1.5). This data is simply relayed by the ISIMaster

- wake up from sleep mode

5   12.1.2.3.4   ISISlave to ISIMaster communication

The ISISlave may wish to communicate the following information to the ISIMaster:

- All data originating from the ISISlave and destined for the external host (see section 12.1.2.1.6). This data is simply relayed by the ISIMaster

12.1.2.3.5   PrintMaster to ISISlave communication

10   When the PrintMaster is not the ISIMaster all ISI communication is done in response to ISI ping packets (see 12.4.4.6). When the PrintMaster is the ISIMaster then it will of course communicate directly with the ISISlaves. The PrintMaster SoPEC may wish to communicate the following information to the ISISlaves:

- Ink status e.g. requests for *dotCount* data i.e. the number of dots in each color fired by the
15   printheads connected to the ISISlaves

- configuration of GPIO ports e.g. for clutch control and lid open detect

- power down command telling the ISISlave to enter sleep mode

- ink cartridge fail information

This list is not complete and the time constraints associated with these requirements have yet to be
20   determined.

In general the PrintMaster may need to be able to:

- send messages to an ISISlave which will cause the ISISlave to return the contents of ISISlave registers to the PrintMaster or

- to program ISISlave registers with values sent by the PrintMaster

25   This should be under the control of software running on the CPU which writes messages to the ISI/SCB interface.

12.1.2.3.6   ISISlave to PrintMaster communication

ISISlaves may need to communicate the following information to the PrintMaster:

- ink status e.g. *dotCount* data i.e. the number of dots in each color fired by the printheads connected to
30   the ISISlaves

- band related information e.g. finished band interrupts

- page related information i.e. buffer underrun, page finished interrupts

- MMU security violation interrupts

- GPIO interrupts and status e.g. clutch control and lid open detect

35   - printhead temperature

- printhead dead nozzle information from SoPEC printhead nozzle tests

- power management status

This list is not complete and the time constraints associated with these requirements have yet to be determined.

As the ISI is an insecure interface commands issued over the ISI should be of limited capability e.g. only limited register writes allowed. The software protocol needs to be constructed with this in mind. In general ISISlaves may need to return register or status messages to the PrintMaster or ISIMaster. They may also need to indicate to the PrintMaster or ISIMaster that a particular interrupt has occurred on the ISISlave. This should be under the control of software running on the CPU which writes messages to the ISI block.

12.1.2.3.7   ISISlave to ISISlave communication

The amount of information that will need to be communicated between ISISlaves will vary considerably depending on the printer configuration. In some systems ISISlave devices will only need to exchange small amounts of control information with each other while in other systems (such as those employing a storage SoPEC or extra USB connection) large amounts of compressed page data may be moved between ISISlaves. Scenarios where ISISlave to ISISlave communication is required include: (a) when the PrintMaster is not the ISIMaster, (b) QA Chip ink usage protocols, (c) data transmission from data storage SoPECs, (d) when there are multiple external host connections supplying data to the printer.

12.1.3   SCB Block Diagram

The SCB consists of four main sub-blocks, as shown in the basic block diagram of Figure 28.

12.1.4   Definitions of I/Os

The toplevel I/Os of the SCB are listed in Table 32. A more detailed description of their functionality will be given in the relevant sub-block sections.

Table 32. SCB I/O

Port name	s	I/O	Description
Clocks and Resets			
prst_n	1	In	System reset signal. Active low.
Pclk	1	In	System clock.
usbclk	1	In	48MHz clock for the USB device and host cores. The cores also require a 12MHz clock, which will be generated locally by dividing the 48MHz clock by 4.
isi_cpr_reset_n	1	Out	Signal from the ISI indicating that ISI activity has been detected while in sleep mode and so the chip should be reset. Active low.
usbd_cpr_reset_n	1	Out	Signal from the USB device that a USB reset has occurred. Active low.
USB device IO transceiver signals			
usbd_ts	1	Out	USB device IO transceiver (BUSB2_PM) driver three-state control. Active high enable.

usbd_a	1	Out	USB device IO transceiver (BUSB2_PM) driver data input.
usbd_se0	1	Out	USB device IO transceiver (BUSB2_PM) single-ended zero input. Active high.
usbd_zp	1	In	USB device IO transceiver (BUSB2_PM) D+ receiver output.
usbd_zm	1	In	USB device IO transceiver (BUSB2_PM) D- receiver output.
usbd_z	1	In	USB device IO transceiver (BUSB2_PM) differential receiver output.
usbd_pull_up_en	1	Out	USB device pull-up resistor enable. Switches power to the external pull-up resistor, connected to the D+ line that is required for device identification to the USB. Active high.
usbd_vbus_sense	1	In	USB device VBUS power sense. Used to detect power on VBUS. NOTE: The IBM Cu11 PADS are 3.3V, VBUS is 5V. An external volt-age conversion will be necessary, e.g. resistor divider network. Active high.
USB host IO transceiver signals			
usbh_ts	1	Out	USB host IO transceiver (BUSB2_PM) driver three-state control. Active high enable
usbh_a	1	Out	USB host IO transceiver (BUSB2_PM) driver data input.
usbh_se0	1	Out	USB host IO transceiver (BUSB2_PM) single-ended zero input. Active high.
usbh_zp	1	In	USB host IO transceiver (BUSB2_PM) D+ receiver output.
usbh_zm	1	In	USB host IO transceiver (BUSB2_PM) D- receiver output.
usbh_z	1	In	USB host IO transceiver (BUSB2_PM) differential receiver output.
usbh_over_current	1	In	USB host port power over current indicator. Active high.
usbh_power_en	1	Out	USB host VBUS power enable. Used for port power switching. Active high.
CPU Interface			
cpu_adr[n:2]	n-1	In	CPU address bus.

cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
scb_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
cpu_scb_sel	1	In	Block select from the CPU. When *cpu_scb_sel* is high both *cpu_adr* and *cpu_dataout* are valid
scb_cpu_rdy	1	Out	Ready signal to the CPU. When *scb_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the SCB and for a read cycle this means the data on *scb_cpu_data* is valid.
scb_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
scb_cpu_debug_valid	1	Out	Signal indicating that the data currently on *scb_cpu_data* is valid debug data
Interrupt signals			
dma_icu_irq	1	Out	DMA interrupt signal to the interrupt controller block.
isi_icu_irq	1	Out	ISI interrupt signal to the interrupt controller block.
usb_icu_irq[1:0]	2	Out	USB host and device interrupt signals to the ICU. Bit 0 - USB Host interrupt Bit 1 - USB Device interrupt
DIU interface			
scb_diu_wadr[21:5]	17	Out	Write address bus to the DIU
scb_diu_data[63:0]	64	Out	Data bus to the DIU.
scb_diu_wreq	1	Out	Write request to the DIU
diu_scb_wack	1	In	Acknowledge from the DIU that the write request was accepted.
scb_diu_wvalid	1	Out	Signal from the SCB to the DIU indicating that the data currently on the *scb_diu_data[63:0]* bus is valid

scb_diu_wmask[7:0]	7	Out	Byte aligned write mask. A "1" in a bit field of "*scb_diu_wmask[7:0]*" means that the corresponding byte will be written to DRAM.
scb_diu_rreq	1	Out	Read request to the DIU.
scb_diu_radr[21:5]	17	Out	Read address bus to the DIU
diu_scb_rack	1	In	Acknowledge from the DIU that the read request was accepted.
diu_scb_rvalid	1	In	Signal from the DIU to the SCB indicating that the data currently on the *diu_data[63:0]* bus is valid
diu_data[63:0]	64	In	Common DIU data bus.
GPIO interface			
isi_gpio_dout[3:0]	4	Out	ISI output data to GPIO pins
isi_gpio_e[3:0]	4	Out	ISI output enable to GPIO pins
gpio_isi_din[3:0]	4	In	Input data from GPIO pins to ISI

12.1.5 SCB Data Flow

A logical view of the SCB is shown in Figure 29, depicting the transfer of data within the SCB.

12.2 USBD (USB DEVICE SUB-BLOCK)

12.2.1 Overview

The FS USB device controller core and associated SCB logic are referred to as the USB Device (USBD).

A SoPEC printer has FS USB device capability to facilitate communication between an external USB host and a SoPEC printer. The USBD is self-powered. It connects to an external USB host via a dedicated USB interface on the SoPEC printer, comprising a USB connector, the necessary discretes for USB signalling and the associated SoPEC ASIC I/Os.

The FS USB device core will be third party IP from Synopsys: TymeWare™ USB1.1 Device Controller (UDCVCI). Refer to the UDCVCI User Manual [20] for a description of the core.

The device core does not support LS USB operation. Control and bulk transfers are supported by the device. Interrupt transfers are not considered necessary because the required interrupt-type functionality can be achieved by sending query messages over the control channel on a scheduled basis. There is no requirement to support isochronous transfers.

The device core is configured to support 6 USB endpoints (EPs): the default control EP (EP0), 4 bulk OUT EPs (EP1, EP2, EP3, EP4) and 1 bulk IN EP (EP5). It should be noted that the direction of each EP is with respect to the USB host, i.e. *IN* refers to data transferred to the external host and *OUT* refers to data transferred from the external host. The 4 bulk OUT EPs will be used for the transfer of data from the external host to SoPEC, e.g. compressed page data, program data or control messages. Each bulk OUT EP can be mapped on to any target destination in a multi-SoPEC system, via the SCB Map configuration registers. The bulk IN EP is used for the transfer of data from SoPEC to the external host, e.g. a print image downloaded from a digital camera that requires

processing on the external host system. Any feedback data will be returned to the external host on EP0, e.g. status information.

The device core does not provide internal buffering for any of its EPs (with the exception of the 8 byte setup data payload for control transfers). All EP buffers are provided in the SCB. Buffers will be grouped according to EP direction and associated packet destination. The SCB Map configuration registers contain a *DestISIId* and *DestISISubId* for each OUT EP, defining their EP mapping and therefore their packet destination. Refer to section Section 12.4 ISI (Inter SoPEC Interface Sub-block) for further details on *ISIId* and *ISISubId*. Refer to section Section 12.5 CTRL (Control Sub-block) for further details on the mapping of OUT EPs.

12.2.2   USBD effective bandwidth

The effective bandwidth between an external USB host and the printer will be influenced by:

- Amount of activity from other devices that share the USB with the printer.
- Throughput of the device controller core.
- EP buffering implementation.
- Responsiveness of the external host system CPU in handling USB interrupts.

To maximize bandwidth to the printer it is recommended that no other devices are active on the USB between the printer and the external host. If the printer is connected to a HS USB external host or hub it may limit the bandwidth available to other devices connected to the same hub but it would not significantly affect the bandwidth available to other devices upstream of the hub. The EP buffering should not limit the USB device core throughput, under normal operating conditions.

Used in the recommended configuration, under ideal operating conditions, it is expected that an effective bandwidth of 8-9 Mbit/s will be achieved with bulk transfers between the external host and the printer.

12.2.3   IN EP packet buffer

The IN EP packet buffer stores packets originating from the LEON CPU that are destined for transmission over the USB to the external USB host. CPU writes to the buffer are 32 bits wide. USB device core reads from the buffer 32 bits wide.

128 bytes of local memory are required in total for EP0-IN and EP5-IN buffering. The IN EP buffer is a single, 2-port local memory instance, with a dedicated read port and a dedicated write port. Both ports are 32 bits wide. Each IN EP has a dedicated 64 byte packet location available in the memory array to buffer a single USB packet (maximum USB packet size is 64 bytes). Each individual 64 byte packet location is structured as 16 x 32 bit words and is read/written in a FIFO manner. When the device core reads a packet entry from the IN EP packet buffer, the buffer must retain the packet until the device core performs a status write, informing the SCB that the packet has been accepted by the external USB host and can be flushed. The CPU can therefore only write a single packet at a time to each IN EP. Any subsequent CPU write request to a buffer location containing a valid packet will be refused, until that packet has been successfully transmitted.

12.2.4   OUT EP packet buffer

The OUT EP packet buffer stores packets originating from the external USB host that are destined for transmission over DMAChannel0, DMAChannel1 or the ISI. The SCB control logic is responsible

114

for routing the OUT EP packets from the OUT EP packet buffer to DMA or to the ISITx Buffer, based on the SCB Map configuration register settings. USB core writes to the buffer are 32 bits wide. DMA and ISI associated reads from the buffer are both 64 bits wide.

512 bytes of local memory are required in total for EP0-OUT, EP1-OUT, EP2-OUT, EP3-OUT and EP4-OUT buffering. The OUT EP packet buffer is a single, 2-port local memory instance, with a dedicated read port and a dedicated write port. Both ports are 64 bits wide. Byte enables are used for the 32 bit wide USB device core writes to the buffer. Each OUT EP can be mapped to DMAChannel0, DMAChannel1 or the ISI.

The OUT EP packet buffer is partitioned accordingly, resulting in three distinct packet FIFOs:

- USBDDMA0FIFO, for USB packets destined for DMAChannel0 on the local SoPEC.
- USBDDMA1FIFO, for USB packets destined for DMAChannel1 on the local SoPEC.
- USBDISIFIFO, for USB packets destined for transmission over the ISI.

12.2.4.1 USBDDMAnFIFO

This description applies to USBDDMA0FIFO and USBDDMA1FIFO, where 'n' represents the respective DMA channel, i.e. $n=0$ for USBDDMA0FIFO, $n=1$ for USBDDMA1FIFO.

USBDDMAnFIFO services any EPs mapped to DMAChannel$n$ on the local SoPEC device. This implies that a packet originating from an EP with an associated *ISIId* that matches the local SoPEC *ISIId* and an *ISISubId=n* will be written to USBDDMAnFIFO, if there is space available for that packet.

USBDDMAnFIFO has a capacity of 2 x 64 byte packet entries, and can therefore buffer up to 2 USB packets. It can be considered as a 2 packet entry FIFO. Packets will be read from it in the same order in which they were written, i.e. the first packet written will be the first packet read and the second packet written will be the second packet read. Each individual 64 byte packet location is structured as 8 x 64 bit words and is read/written in a FIFO manner.

The USBDDMAnFIFO has a write granularity of 64 bytes, to allow for the maximum USB packet size. The USBDDMAnFIFO will have a read granularity of 32 bytes to allow for the DMA write access bursts of 4 x 64 bit words, i.e. the DMA Manager will read 32 byte chunks at a time from the USBDDMAnFIFO 64byte packet entries, for transfer to the DIU.

It is conceivable that a packet which is not a multiple 32 bytes in size may be written to the USBDDMAnFIFO. When this event occurs, the DMA Manager will read the contents of the remaining address locations associated with the 32 byte chunk in the USBDDMAnFIFO, transferring the packet plus whatever data is present in those locations, resulting in a 32 byte packet (a burst of 4 x 64 bit words) transfer to the *DIU*.

The DMA channels should achieve an effective bandwidth of 160 Mbits/sec (1 bit/cycle) and should never become blocked, under normal operating conditions. As the USB bandwidth is considerably less, a 2 entry packet FIFO for each DMA channel should be sufficient.

12.2.4.2 USBDISIFIFO

USBDISIFIFO services any EPs mapped to ISI. This implies that a packet originating from an EP with an associated *ISIId* that does not match the local SoPEC *ISIId* will be written to USBDISIFIFO if there is space available for that packet.

USBDISIFIFO has a capacity of 4 x 64 byte packet entries, and can therefore buffer up to 4 USB packets. It can be considered as a 4 packet entry FIFO. Packets will be read from it in the same order in which they were written, i.e. the first packet written will be the first packet read and the second packet written will be the second packet read, etc. Each individual 64 byte packet location is structured as 8 x 64 bit words and is read/written in a FIFO manner.

The ISI long packet format will be used to transfer data across the ISI. Each ISI long packet data payload is 32 bytes. The USBDISIFIFO has a write granularity of 64 bytes, to allow for the maximum USB packet size. The USBDISIFIFO will have a read granularity of 32 bytes to allow for the ISI packet size, i.e. the SCB will read 32 byte chunks at a time from the USBDISIFIFO 64byte packet entries, for transfer to the ISI.

It is conceivable that a packet which is not a multiple 32 bytes in size may be written to the USBDISIFIFO, either intentionally or due to a software error. A maskable interrupt per EP is provided to flag this event. There will be 2 options for dealing with this scenario on a per EP basis:

- Discard the packet.
- Read the contents of the remaining address locations associated with the 32 byte chunk in the USBDISIFIFO, transferring the irregular size packet plus whatever data is present in those locations, resulting in a 32 byte packet transfer to the *ISITxBuffer*.

The ISI should achieve an effective bandwidth of 100 Mbits/sec (4 wire configuration). It is possible to encounter a number of retries when transmitting an ISI packet and the LEON CPU will require access to the ISI transmit buffer. However, considering the relatively low bandwidth of the USB, a 4 packet entry FIFO should be sufficient.

12.2.5   Wake-up from sleep mode

The SoPEC will be placed in sleep mode after a suspend command is received by the USB device core. The USB device core will continue to be powered and clocked in sleep mode. A USB reset, as opposed to a device resume, will be required to bring SoPEC out of its sleep state as the sleep state is hoped to be logically equivalent to the power down state.

The USB reset signal originating from the USB controller will be propagated to the CPR (as *usb_cpr_reset_n*) if the *USBWakeupEnable* bit of the *WakeupEnable* register (see Table   ) has been set. The *USBWakeupEnable* bit should therefore be set just prior to entering sleep mode.

There is a scenario that would require SoPEC to initiate a USB remote wake-up (i.e. where SoPEC signals resume to the external USB host after being suspended by the external USB host). A digital camera (or other supported external USB device) could be connected to SoPEC via the internal SoPEC USB host controller core interface. There may be a need to transfer data from this external USB device, via SoPEC, to the external USB host system for processing. If the USB connecting the external host system and SoPEC was suspended, then SoPEC would need to initiate a USB remote wake-up.

12.2.6   Implementation

12.2.6.1 USBD Sub-block Partition

                * Block diagram
                * Definition of I/Os

12.2.6.2 USB Device IP Core

12.2.6.3 PVCI Target

12.2.6.4 IN EP Buffer

12.2.6.5 OUT EP Buffer

12.3   USBH (USB Host Sub-block)

12.3.1   Overview

The SoPEC USB Host Controller (HC) core, associated SCB logic and associated SoPEC ASIC I/Os are referred to as the USB Host (USBH).

A SoPEC printer has FS USB host capability, to facilitate communication between an external USB device and a SoPEC printer. The USBH connects to an external USB device via a dedicated USB interface on the SoPEC printer, comprising a USB connector, the necessary discretes for USB signalling and the associated SoPEC ASIC I/Os.

The FS USB HC core are third party IP from Synopsys: DesignWare[R] USB1.1 OHCI Host Controller with PVCI (UHOSTC_PVCI). Refer to the UHOSTC_PVCI User Manual [18] for details of the core. Refer to the Open Host Controller Interface (OHCI) Specification Release [19] for details of OHCI operation.

The HC core supports Low-Speed (LS) USB devices, although compatible external USB devices are most likely to be FS devices. It is expected that communication between an external USB device and a SoPEC printer will be achieved with control and bulk transfers. However, isochronous and interrupt transfers are also supported by the HC core.

There will be 2 communication channels between the Host Controller Driver (HCD) software running on the LEON CPU and the HC core:

- OHCI operational registers in the HC core. These registers are control, status, list pointers and a pointer to the Host Controller Communications Area (HCCA) in shared memory. A target Peripheral Virtual Component Interface (PCVI) on the HC core will provide LEON with direct read/write access to the operational registers. Refer to the OHCI Specification for details of these registers.

- HCCA in SoPEC eDRAM. An initiator Peripheral Virtual Component Interface (PCVI) on the HC core will provide the HC with DMA read/write access to an address space in eDRAM. The HCD running on LEON will have read/write access to the same address space. Refer to the OHCI Specification for details of the HCCA.

The target PVCI interface is a 32 bit word aligned interface, with byte enables for write access. All read/ write access to the target PVCI interface by the LEON CPU will be 32 bit word aligned. The byte enables will not be used, as all registers will be read and written as 32 bit words.

The initiator PVCI interface is a 32 bit word aligned interface with byte enables for write access. All DMA read/write accesses are 256 bit word aligned, in bursts of 4 x 64 bit words. As there is no guarantee that the read/write requests from the HC core will start at a 256 bit boundary or be 256 bits long, it is necessary to provide 8 byte enables for each of the 64 bit words in a write burst form the HC core to DMA. The signal *scb_diu_wmask* serves this purpose.

Configuration of the HC core will be performed by the HCD.

12.3.2 Read/Write Buffering

The HC core maximum burst size for a read/write access is 4 x 32 bit words. This implies that the minimum buffering requirements for the HC core will be a 1 entry deep address register and a 4 entry deep data register. It will be necessary to provide data and address mapping functionality to convert the 4 x 32 bit word HC core read/write bursts into 4 x 64 bit word DMA read/write bursts. This will meet the minimum buffering requirements.

12.3.3 USBH effective bandwidth

The effective bandwidth between an external USB device and a SoPEC printer will be influenced by:

- Amount of activity from other devices that share the USB with the external USB device.
- Throughput of the HC core.
- HC read/write buffering implementation.
- Responsiveness of the LEON CPU in handling USB interrupts.

Effective bandwidth between an external USB device and a SoPEC printer is not an issue. The primary application of this connectivity is the download of a print image from a digital camera. Printing speed is not important for this type of print operation. However, to maximize bandwidth to the printer it is recommended that no other devices are active on the USB between the printer and the external USB device. The HC read/write buffering in the SCB should not limit the USB HC core throughput, under normal operating conditions.

Used in the recommended configuration, under ideal operating conditions, it is expected that an effective bandwidth of 8-9 Mbit/s will be achieved with bulk transfers between the external USB device and the SoPEC printer.

12.3.4 Implementation

12.3.5 USBH Sub-block Partition

* USBH Block Diagram

* Definition of I/Os.

12.3.5.1 USB Host IP Core

12.3.5.2 PVCI Target

12.3.5.3 PVCI Initiator

12.3.5.4 Read/Write Buffer

12.4 ISI (INTER SOPEC INTERFACE SUB-BLOCK)

12.4.1 Overview

The ISI is utilised in all system configurations requiring more than one SoPEC. An example of such a system which requires four SoPECs for duplex A3 printing and an additional SoPEC used as a storage device is shown in Figure 27.

The ISI performs much the same function between an ISISlave SoPEC and the ISIMaster as the USB connection performs between the ISIMaster and the external host. This includes the transfer of all program data, compressed page data and message (i.e. commands or status information) passing between the ISIMaster and the ISISlave SoPECs. The ISIMaster initiates all communication with the ISISlaves.

12.4.2   ISI Effective Bandwidth

The ISI will need to run at a speed that will allow error free transmission on the PCB while minimising the buffering and hardware requirements on SoPEC. While an ISI speed of 10 Mbit/s is adequate to match the effective FS USB bandwidth it would limit the system performance when a high-speed connection (e.g. USB2.0, IEEE1394) is used to attach the printer to the PC. Although they would require the use of an extra ISI-Bridge chip such systems are envisaged for more expensive printers (compared to the low-cost basic SoPEC powered printers that are initially being targeted) in the future.

An ISI line speed (i.e. the speed of each individual ISI wire) of 32 Mbit/s is therefore proposed as it will allow ISI data to be over-sampled 5 times (at a *pclk* frequency of 160MHz). The total bandwidth of the ISI will depend on the number of pins used to implement the interface. The ISI protocol will work equally well if 2 or 4 pins are used for transmission/reception. The *ISINumPins* register is used to select between a 2 or 4 wire ISI, giving peak raw bandwidths of 64 Mbit/s and 128 Mbit/s respectively. Using either a 2 or 4 wire ISI solution would allow the movement of data in to and out of a storage SoPEC (as described in 12.1.1.4 above), which is the most bandwidth hungry ISI use, in a timely fashion.

The *ISINumPins* register is used to select between a 2 or 4 wire ISI. A 2 wire ISI is the default setting for *ISINumPins* and this may be changed to a 4 wire ISI after initial communication has been established between the ISIMaster and all ISISlaves. Software needs to ensure that the switch from 2 to 4 wires is handled in a controlled and coordinated fashion so that nothing is transmitted on the ISI during the switch over period.

The maximum effective bandwidth of a two wire ISI, after allowing for protocol overheads and bus turnaround times, is expected to be approx. 50 Mbit/s.

12.4.3   ISI Device Identification and Enumeration

The *ISIMasterSel* bit of the *ISICntrl* register (see section Table   ) determines whether a SoPEC is an ISIMaster (*ISIMasterSel* = 1), or an ISISlave (*ISIMasterSel* = 0).

SoPEC defaults to being an ISISlave (*ISIMasterSel* = 0) after a power-on reset - i.e. it will not transmit data on the ISI without first receiving a ping. If a SoPEC's *ISIMasterSel* bit is changed to 1, then that SoPEC will become the ISIMaster, transmitting data without requiring a ping, and generating pings as appropriately programmed.

ISIMasterSel can be set to 1 explicitly by the CPU writing directly to the *ISICntrl* register. *ISIMasterSel* can also be automatically set to 1 when activity occurs on any of USB endpoints 2-4 and the *AutoMasterEnable* bit of the *ISICntrl* register is also 1 (the default reset condition). Note that if *AutoMasterEnable* is 0, then activity on USB endpoints 2-4 will not result in *ISIMasterSel* being set to 1. USB endpoints 2-4 are chosen for the automatic detection since the power-on-reset condition has USB endpoints 0 and 1 pointing to ISIId 0 (which matches the local SoPEC's ISIId after power-on reset). Thus any transmission on USB endpoints 2-4 indicate a desire to transmit on the ISI which would usually indicate ISIMaster status. The automatic setting of *ISIMasterSel* can be disabled by clearing *AutoMasterEnable*, thereby allowing the SoPEC to remain an ISISlave while still making use of the USB endpoints 2-4 as external destinations.

Thus the setting of a SoPEC being ISIMaster or ISISlave can be completely under software control, or can be completely automatic.

The ISIId is established by software downloaded over the ISI (in broadcast mode) which looks at the input levels on a number of GPIO pins to determine the ISIId. For any given printer that uses a multi-SoPEC configuration it is expected that there will always be enough free GPIO pins on the ISISlaves to support this enumeration mechanism.

12.4.4   ISI protocol

The ISI is a serial interface utilizing a 2/4 wire half-duplex configuration such as the 2-wire system shown in Figure 30 below. An ISIMaster must always be present and a variable number of ISISlaves may also be on the ISI bus. The ISI protocol supports up to 14 addressable slaves, however to simplify electrical issues the ISI drivers need only allow for 5-6 ISI devices on a particular ISI bus. The ISI bus enables broadcasting of data, ISIMaster to ISISlave communication, ISISlave to ISIMaster communication and ISISlave to ISISlave communication. Flow control, error detection and retransmission of errored packets is also supported. ISI transmission is asynchronous and a *Start* field is present in every transmitted packet to ensure synchronization for the duration of the packet.

To maximize the effective ISI bandwidth while minimising pin requirements a half-duplex interleaved transmission scheme is used. Figure 31 below shows how a 16-bit word is transmitted from an ISIMaster to an ISISlave over a 2-wire ISI bus. Since data will be interleaved over the wires and a 4-wire ISI is also supported, all ISI packets should be a multiple of 4 bits.

All ISI transactions are initiated by the ISIMaster and every non-broadcast data packet needs to be acknowledged by the addressed recipient. An ISISlave may only transmit when it receives a ping packet (see section 12.4.4.6) addressed to it. To avoid bus contention all ISI devices must wait *ISITurnAround* bit-times (5 *pclk* cycles per bit) after detecting the end of a packet before transmitting a packet (assuming they are required to transmit). All non-transmitting ISI devices must tristate their Tx drivers to avoid line contention. The ISI protocol is defined to avoid devices driving out of order (e.g. when an ISISlave is no longer being addressed). As the ISI uses standard I/O pads there is no physical collision detection mechanism.

There are three types of ISI packet: a long packet (used for data transmission), a ping packet (used by the ISIMaster to prompt ISISlaves for packets) and a short packet (used to acknowledge receipt of a packet). All ISI packets are delineated by a *Start* and *Stop* fields and transmission is atomic i.e. an ISI packet may not be split or halted once transmission has started.

12.4.4.1   ISI transactions

The different types of ISI transactions are outlined in Figure 32 below. As described later all NAKs are inferred and ACKs are not addressed to any particular ISI device.

12.4.4.2 Start Field Description

The *Start* field serves two purposes: To allow the start of a packet be unambiguously identified and to allow the receiving device synchronise to the data stream. The symbol, or data value, used to identify a *Start* field must not legitimately occur in the ensuing packet. Bit stuffing is used to guarantee that the *Start* symbol will be unique in any valid (i.e. error free) packet. The ISI needs to

see a valid Start symbol before packet reception can commence i.e. the receive logic constantly looks for a *Start* symbol in the incoming data and will reject all data until it sees a *Start* symbol. Furthermore if a *Start* symbol occurs (incorrectly) during a data packet it will be treated as the start of a new packet. In this case the partially received packet will be discarded.

5   The data value of the *Start* symbol should guarantee that an adequate number of transitions occur on the physical ISI lines to allow the receiving ISI device to determine the best sampling window for the transmitted data. The *Start* symbol should also be sufficiently long to ensure that the bit stuffing overhead is low but should still be short enough to reduce its own contribution to the packet overhead. A *Start* symbol of b01010101 is therefore used as it is an effective compromise between

10   these constraints.

Each SoPEC in a multi-SoPEC system will derive its system clock from a unique (i.e. one per SoPEC) crystal. The system clocks of each device will drift relative to each other over any period of time. The system clocks are used for generation and sampling of the ISI data. Therefore the sampling window can drift and could result in incorrect data values being sampled at a later point in

15   time. To overcome this problem the ISI receive circuitry tracks the sampling window against the incoming data to ensure that the data is sampled in the centre of the bit period.

12.4.4.3 Stop Field Description

A 1 bit-time *Stop* field of b1 per ISI line ensures that all ISI lines return to the high state before the next packet is transmitted. The stop field is driven on to each ISI line simultaneously, i.e. b11 for a

20   2-wire ISI and b1111 for a 4-wire ISI would be interleaved over the respective ISI lines. Each ISI line is driven high for 1 bit-time. This is necessary because the first bit of the *Start* field is b0.

12.4.4.4 Bit Stuffing

This involves the insertion of bits into the bitstream at the transmitting SoPEC to avoid certain data patterns. The receiving SoPEC will strip these inserted bits from the bitstream.

25   Bit-stuffing will be performed when the *Start* symbol appears at a location other than the start field of any packet, i.e. when the bit pattern b0101010 occurs at the transmitter, a 0 will be inserted to escape the *Start* symbol, resulting in the bit pattern b01010100. Conversely, when the bit pattern b0101010 occurs at the receiver, if the next bit is a '0' it will be stripped, if it is a '1' then a *Start* symbol is detected.

30   If the frequency variations in the quartz crystal were large enough, it is conceivable that the resultant frequency drift over a large number of consecutive 1s or 0s could cause the receiving SoPEC to loose synchronisation.[6] The quartz crystal that will be used in SoPEC systems is rated for 32MHz @ 100ppm. In a multi-SoPEC system with a 32MHz+100ppm crystal and a 32MHz-100ppm crystal, it would take approximately 5000 *pclk* cycles to cause a drift of 1 pclk cycle. This means that we would only

35   need to bit-stuff somewhere before 1000 ISI bits of consecutive 1s or consecutive 0s, to ensure adequate

---

[6]Current max packet size ~= 290 bits = 145 bits per ISI line (on a 2 wire ISI) = 725 160MHz cycles. Thus the *pclks* in the two communicating ISI devices should not drift by more than one cycle in 725 i.e. 1379 ppm. Careful analysis of the crystal, PLL and oscillator specs and the sync detection circuit is needed here to ensure our solution is robust.

synchronization. As the maximum number of bits transmitted per ISI line in a packet is 145, it should not be necessary to perform bit-stuffing for consecutive 1s or 0s. We may wish to constrain the spec of *xtalin* and also *xtalin* for the ISI-Bridge chip to ensure the ISI cannot drift out of sync during packet reception.

Note that any violation of bit stuffing will result in the *RxFrameErrorSticky* status bit being set and the incoming packet will be treated as an errored packet.

.*12.4.4.5 ISI Long Packet*

The format of a long ISI packet is shown in Figure 33 below. Data may only be transferred between ISI devices using a long packet as both the short and ping packets have no payload field. Except in the case of a broadcast packet, the receiving ISI device will always reply to a long packet with an explicit ACK (if no error is detected in the received packet) or will not reply at all (e.g. an error is detected in the received packet), leaving the transmitter to infer a NAK. As with all ISI packets the bitstream of a long packet is transmitted with its lsb (the leftmost bit in Figure 33) first. Note that the total length (in bits) of an ISI long packet differs slightly between a 2 and 4-wire ISI system due to the different number of bits required for the *Start* and *Stop* fields.

All long packets begin with the *Start* field as described earlier. The *PktDesc* field is described in Table 33.

Table 33. PktDesc field description

Bit	Description
0:1	00 - Long packet 01 - Reserved 10 - Ping packet 11 – Reserved
2	Sequence bit value. Only valid for long packets. See section 12.4.4.9 for a description of sequence bit operation

Any ISI device in the system may transmit a long packet but only the ISIMaster may initiate an ISI transaction using a long packet. An ISISlave may only send a long packet in reply to a ping message from the ISIMaster. A long packet from an ISISlave may be addressed to any ISI device in the system.

The *Address* field is straightforward and complies with the ISI naming convention described in section 12.5.

The payload field is exactly what is in the transmit buffer of the transmitting ISI device and gets copied into the receive buffer of the addressed ISI device(s). When present the payload field is always 256 bits.

To ensure strong error detection a 16-bit CRC is appended.

12.4.4.6 ISI Ping Packet

The ISI ping packet is used to allow ISISlaves to transmit on the ISI bus. As can be seen from Figure 34 below the ping packet can be viewed as a special case of the long packet. In other words it is a long packet without any payload. Therefore the *PktDesc* field is the same as a long packet *PktDesc*, with the exception of the sequence bit, which is not valid for a ping packet. Both the *ISISubId* and the sequence bit are fixed at 1 for all ping packets. These values were chosen to

maximize the hamming distance from an ACK symbol and to minimize the likelihood of bit stuffing. The *ISISubId* is unused in ping packets because the ISIMaster is addressing the ISI device rather than one of the DMA channels in the device. The ISISlave may address any *ISIId.ISISubId* in response if it wishes. The ISISlave will respond to a ping packet with either an explicit ACK (if it has

5   nothing to send), an inferred NAK (if it detected an error in the ping packet) or a long packet (containing the data it wishes to send). Note that inferred NAKs do not result in the retransmission of a ping packet. This is because the ping packet will be retransmitted on a predetermined schedule (see 12.4.4.11 for more details).

An ISISlave should never respond to a ping message to the broadcast *ISIId* as this must have been

10   sent in error. An ISI ping packet will never be sent in response to any packet and may only originate from an ISIMaster.

12.4.4.7 ISI Short Packet

The ISI short packet is only 17 bits long, including the *Start* and *Stop* fields. A value of b11101011 is proposed for the ACK symbol. As a 16-bit CRC is inappropriate for such a short packet it is not

15   used. In fact there is only one valid value for a short ACK packet as the *Start*, ACK and *Stop* symbols all have fixed values. Short packets are only used for acknowledgements (i.e. explicit ACKs). The format of a short ISI packet is shown in Figure 35 below. The ACK value is chosen to ensure that no bit stuffing is required in the packet and to minimize its hamming distance from ping and long ISI packets.

20

12.4.4.8 Error Detection and Retransmission

The 16-bit CRC will provide a high degree of error detection and the probability of transmission errors occurring is very low as the transmission channel (i.e. PCB traces) will have a low inherent bit error rate. The number of undetected errors should therefore be minute.

25   The HDLC standard CRC-16 (i.e. $G(x) = x^{16} + x^{12} + x^5 + 1$) is to be used for this calculation, which is to be performed serially. It is calculated over the entire packet (excluding the *Start* and *Stop* fields). A simple retransmission mechanism frees the CPU from getting involved in error recovery for most errors because the probability of a transmission error occurring more than once in succession is very, very low in normal circumstances.

30   After each non-short ISI packet is transmitted the transmitting device will open a reply window. The size of the reply window will be *ISIShortReplyWin* bit times when a short packet is expected in reply, i.e. the size of a short packet, allowing for worst case bit stuffing, bus turnarounds and timing differences. The size of the reply window will be *ISILongReplyWin* bit times when a long packet is expected in reply, i.e. this will be the max size of a long packet, allowing for worst case bit stuffing,

35   bus turnarounds and timing differences. In both cases if an ACK is received the window will close and another packet can be transmitted but if an ACK is not received then the full length of the window must be waited out.

As no reply should be sent to a broadcast packet, no reply window should be required however all other long packets open a reply window in anticipation of an ACK. While the desire is to minimize

the time between broadcast transmissions the simplest solution should be employed. This would imply the same size reply window as other long packets.

When a packet has been received without any errors the receiving ISI device must transmit its acknowledge packet (which may be either a long or short packet) before the reply window closes.

5   When detected errors do occur the receiving ISI device will not send any response. The transmitting ISI device interprets this lack of response as a NAK indicating that errors were detected in the transmitted packet or that the receiving device was unable to receive the packet for some reason (e.g. its buffers are full). If a long packet was transmitted the transmitting ISI device will keep the transmitted packet in its transmit buffer for retransmission. If the transmitting device is the ISIMaster

10   it will retransmit the packet immediately while if the transmitting device is an ISISlave it will retransmit the packet in response to the next ping it receives from the ISIMaster.

The transmitting ISI device will continue retransmitting the packet when it receives a NAK until it either receives an ACK or the number of retransmission attempts equals the value of the *NumRetries* register. If the transmission was unsuccessful then the transmitting device sets the

15   *TxErrorSticky* bit in its *ISIIntStatus* register. The receiving device also sets the *RxErrorSticky* bit in its *ISIIntStatus* register whenever it detects a CRC error in an incoming packet and is not required to take any further action, as it is up to the transmitting device to detect and rectify the problem. The *NumRetries* registers in all ISI devices should be set to the same value for consistent operation. Note that successful transmission or reception of ping packets do not affect retransmission

20   operation.

Note that a transmit error will cause the ISI to stop transmitting. CPU intervention will be required to resolve the source of the problem and to restart the ISI transmit operation. Receive errors however do not affect receive operation and they are collected to facilitate problem debug and to monitor the quality of the ISI physical channel. Transmit or receive errors should be extremely rare and their

25   occurrence will most likely indicate a serious problem.

Note that broadcast packets are never acknowledged to avoid contention on the common ISI lines. If an ISISlave detects an error in a broadcast packet it should use the message passing mechanism described earlier to alert the ISIMaster to the error if it so wishes.

12.4.4.9   Sequence Bit Operation

30   To ensure that communication between transmitting and receiving ISI devices is correctly ordered a sequence bit is included in every long packet to keep both devices in step with each other. The sequence bit field is a constant for short or ping packets as they are not used for data transmission. In addition to the transmitted sequence bit all ISI devices keep two local sequence bits, one for each *ISISubId*. Furthermore each ISI device maintains a transmit sequence bit for each *ISIId* and

35   *ISISubId* it is in communication with. For packets sourced from the external host (via USB) the transmit sequence bit is contained in the relevant *USBEPnDest* register while for packets sourced from the CPU the transmit sequence bit is contained in the *CPUISITxBuffCntrl* register. The sequence bits for received packets are stored in *ISISubId0Seq* and *ISISubId1Seq* registers. All ISI devices will initialize their sequence bits to 0 after reset. It is the responsibility of software to ensure

that the sequence bits of the transmitting and receiving ISI devices are correctly initialized each time a new source is selected for any *ISIId.ISISubId* channel.

Sequence bits are ignored by the receiving ISI device for broadcast packets. However the broadcasting ISI device is free to toggle the sequence in the broadcast packets since they will not affect operation. The SCB will do this for all USB source data so that there is no special treatment for the sequence bit of a broadcast packet in the transmitting device. CPU sourced broadcasts will have sequence bits toggled at the discretion of the program code.

Each SoPEC may also ignore the sequence bit on either of its ISISubId channels by setting the appropriate bit in the *ISISubIdSeqMask* register. The sequence bit should be ignored for ISISubId channels that will carry data that can originate from more than one source and is self ordering e.g. control messages.

A receiving ISI device will toggle its sequence bit addressed by the ISISubId only when the receiver is able to accept data and receives an error-free data packet addressed to it. The transmitting ISI device will toggle its sequence bit for that ISIId.ISISubId channel only when it receives a valid ACK handshake from the addressed ISI device.

Figure 36 shows the transmission of two long packets with the sequence bit in both the transmitting and receiving devices toggling from 0 to 1 and back to 0 again. The toggling operation will continue in this manner in every subsequent transmission until an error condition is encountered.

When the receiving ISI device detects an error in the transmitted long packet or is unable to accept the packet (because of full buffers for example) it will not return any packet and it will not toggle its local sequence bit. An example of this is depicted in Figure 37. The absence of any response prompts the transmitting device to retransmit the original (seq=0) packet. This time the packet is received without any errors (or buffer space may have been freed) so the receiving ISI device toggles its local sequence bit and responds with an ACK. The transmitting device then toggles its local sequence bit to a 1 upon correct receipt of the ACK.

However it is also possible for the ACK packet from the receiving ISI device to be corrupted and this scenario is shown in Figure 38. In this case the receiving device toggles its local sequence bit to 1 when the long packet is received without error and replies with an ACK to the transmitting device. The transmitting device does not receive the ACK correctly and so does not change its local sequence bit. It then retransmits the seq=0 long packet. When the receiving device finds that there is a mismatch between the transmitted sequence bit and the expected (local) sequence bit is discards the long packet and replies with an ACK. When the transmitting ISI device correctly receives the ACK it updates its local sequence bit to a 1, thus restoring synchronization. Note that when the *ISISubIdSeqMask* bit for the addressed ISISubId is set then the retransmitted packet is not discarded and so a duplicate packet will be received. The data contained in the packet should be self-ordering and so the software handling these packets (most likely control messages) is expected to deal with this eventuality.

12.4.4.10 Flow Control

The ISI also supports flow control by treating it in exactly the same manner as an error in the received packet. Because the SCB enjoys greater guaranteed bandwidth to DRAM than both the ISI and USB can supply flow control should not be required during normal operation. Any blockage on a DMA channel will soon result in the *NumRetries* value being exceeded and transmission from that SoPEC being halted. If a SoPEC NAKs a packet because its RxBuffer is full it will flag an overflow condition. This condition can potentially cause a CPU interrupt, if the corresponding interrupt is enabled. The *RxOverflowSticky* bit of its *ISIIntStatus* register reflects this condition. Because flow control is treated in the same manner as an error the transmitting ISI device will not be able to differentiate a flow control condition from an error in the transmitted packet.

12.4.4.11 Auto-ping Operation

While the CPU of the ISIMaster could send a ping packet by writing the appropriate header to the *CPUISITxBuffCntrl* register it is expected that all ping packets will be generated in the ISI itself. The use of automatically generated ping packets ensures that ISISlaves will be given access to the ISI bus with a programmable minimum guaranteed frequency in addition to whenever it would otherwise be idle. Five registers facilitate the automatic generation of ping messages within the ISI: *PingSchedule0, PingSchedule1, PingSchedule2, ISITotalPeriod* and *ISILocalPeriod*. Auto-pinging will be enabled if any bit of any of the *PingScheduleN* registers is set and disabled if all *PingScheduleN* registers are 0x0000.

Each bit of the 15-bit *PingScheduleN* register corresponds to an ISIId that is used in the *Address* field of the ping packet and a 1 in the bit position indicates that a ping packet is to be generated for that ISIId. A 0 in any bit position will ensure that no ping packet is generated for that ISIId. As ISISlaves may differ in their bandwidth requirement (particularly if a storage SoPEC is present) three different PingSchedule registers are used to allow an ISISlave receive up to three times the number of pings as another active ISISlave. When the ISIMaster is not sending long packets (sourced from either the CPU or USB in the case of a SoPEC ISIMaster) ISI ping packets will be transmitted according to the pattern given by the three *PingScheduleN* registers. The ISI will start with the lsb of *PingSchedule0* register and work its way from lsb through msb of each of the *PingScheduleN* registers. When the msb of *PingSchedule2* is reached the ISI returns to the lsb of *PingSchedule0* and continues to cycle through each bit position of each *PingScheduleN* register. The ISI has more than enough time to work out the destination of the next ping packet while a ping or long packet is being transmitted.

With the addition of auto-ping operation we now have three potential sources of packets in an ISIMaster SoPEC: USB, CPU and auto-ping. Arbitration between the CPU and USB for access to the ISI is handled outside the ISI. To ensure that local packets get priority whenever possible and that ping packets can have some guaranteed access to the ISI we use two 4-bit counters whose reload value is contained in the *ISITotalPeriod* and *ISILocalPeriod* registers. As we saw in section 12.4.4.1 every ISI transaction is initiated by the ISIMaster transmitting either a long packet or a ping packet. The *ISITotalPeriod* counter is decremented for every ISI transaction (i.e. either long or ping) when its value is non-zero. The *ISILocalPeriod* counter is decremented for every local packet that is transmitted. Neither counter is decremented by a retransmitted packet. If the *ISITotalPeriod* counter

is zero then ping packets will not change its value from zero. Both the *ISITotalPeriod* and *ISILocalPeriod* counters are reloaded by the next local packet transmit request after the *ISITotalPeriod* counter has reached zero and this local packet has priority over pings.

The amount of guaranteed ISI bandwidth allocated to both local and ping packets is determined by the values of the *ISITotalPeriod* and *ISILocalPeriod* registers. Local packets will always be given priority when the *ISILocalPeriod* counter is non-zero. Ping packets will be given priority when the *ISILocalPeriod* counter is zero and the *ISITotalPeriod* counter is still non-zero.

Note that ping packets are very likely to get more than their guaranteed bandwidth as they will be transmitted whenever the ISI bus would otherwise be idle (i.e. no pending local packets). In particular when the *ISITotalPeriod* counter is zero it will not be reloaded until another local packet is pending and so ping packets transmitted when the *ISITotalPeriod* counter is zero will be in addition to the guaranteed bandwidth. Local packets on the other hand will never get more than their guaranteed bandwidth because each local packet transmitted decrements both counters and will cause the counters to be reloaded when the *ISITotalPeriod* counter is zero. The difference between the values of the *ISITotalPeriod* and *ISILocalPeriod* registers determines the number of automatically generated ping packets that are guaranteed to be transmitted every *ISITotalPeriod* number of ISI transactions. If the *ISITotalPeriod* and *ISILocalPeriod* values are the same then the local packets will always get priority and could totally exclude ping packets if the CPU always has packets to send.

For example if *ISITotalPeriod* = 0xC; *ISILocalPeriod* = 0x8; *PingSchedule0* = 0x0E; *PingSchedule1* = 0x0C and *PingSchedule2* = 0x08 then four ping messages are guaranteed to be sent in every 12 ISI transactions. Furthermore ISIId3 will receive 3 times the number of ping packets as ISId1 and ISIId2 will receive twice as many as ISId1. Thus over a period of 36 contended ISI transactions (allowing for two full rotations through the three *PingScheduleN* registers) when local packets are always pending 24 local packets will be sent, ISId1 will receive 2 ping packets, ISId2 will receive 4 pings and ISId3 will receive 6 ping packets. If local traffic is less frequent then the ping frequency will automatically adjust upwards to consume all remaining ISI bandwidth.

12.4.5   Wake-up from Sleep Mode

Either the PrintMaster SoPEC or the external host may place any of the ISISlave SoPECs in sleep mode prior to going into sleep mode itself. The ISISlave device should then ensure that its *ISIWakeupEnable* bit of the *WakeupEnable* register (see Table 34) is set prior to entering sleep mode. In an ISISlave device the ISI block will continue to receive power and clock during sleep mode so that it may monitor the *gpio_isi_din* lines for activity. When ISI activity is detected during sleep mode and the *ISIWakeupEnable* bit is set the ISI asserts the *isi_cpr_reset_n* signal. This will bring the rest of the chip out of sleep mode by means of a wakeup reset. See chapter 16 for more details of reset propagation.

12.4.6   Implementation

Although the ISI consists of either 2 or 4 ISI data lines over which a serial data stream is demultiplexed, each ISI line is treated as a separate serial link at the physical layer. This permits a certain amount of skew between the ISI lines that could not be tolerated if the lines were treated as

a parallel bus. A lower Bit Error Rate (BER) can be achieved if the serial data recovery is performed separately on each serial link. Figure 39 illustrates the ISI sub block partitioning.

12.4.6.1 ISI Sub-block Partition

* Definition of I/Os.

Table 34. ISI I/O

Port name	Pins	I/O	Description
Clock and Reset			
isi_pclk	1	In	ISI primary clock.
isi_reset_n	1	In	ISI reset. Active low. Asserting *isi_reset_n* will reset all ISI logic. Synchronous to isi_pclk.
Configuration			
isi_go	1	In	ISI GO. Active high. When GO is de-asserted, all ISI statemachines are reset to their idle states, all ISI output signals are de-asserted, but all ISI counters retain their values. When GO is asserted, all ISI counters are reset and all ISI statemachines and output signals will return to their normal mode of operation.
isi_master_select	1	In	ISI master select. Determines whether the SoPEC is an ISIMaster or not 1 = ISIMaster 0 = ISISlave
isi_id[3:0]	4	In	ISI ID for this device.
isi_retries[3:0]	4	In	ISI number of retries. Number of times a transmitting ISI device will attempt retransmission of a NAK'd packet before aborting the transmission and flagging an error. The value of this configuration signal should not be changed while there are valid packets in the Tx buffer.
isi_ping_schedule0[14 :0]	15	In	ISI auto ping schedule #0. Denotes which ISIIds will be receive ping packets. Note that bit0 refers to ISIId0, bit1 to ISIId1...bit14 to ISIId14. Setting a bit in this schedule will enable auto ping generation for the corresponding ISI ID. The ISI will start from the bit 0 of *isi_ping_schedule0* and cycle through to bit 14, generating pings for each bit that is set. This operation will be performed in sequence from

			isi_ping_schedule0 through isi_ping_schedule2.
isi_ping_schedule1[14 :0]	15	In	As per isi_ping_schedule0.
isi_ping_schedule2[14 :0]	15	In	As per isi_ping_schedule0.
isi_total_period[3:0]	4	In	Reload value of the ISI Total Period Counter.
isi_local_period[3:0]	4	In	Reload value of the ISI Local Period Counter.
isi_number_pins	1	In	Number of active ISI data pins. Used to select how many serial data pins will be used to transmit and receive data. Should reflect the number of ISI device data pins that are in use. 1 = isi_data[3:0] active 0 = isi_data[1:0] active
isi_turn_around[3:0]	4	In	ISI bus turn around time in ISI clock cycles (32MHz).
isi_short_reply_win[4:0]	5	In	ISI long packet reply window in ISI clock cycles (32MHz).
isi_long_reply_win[8:0]	9	In	ISI long packet reply window in ISI clock cycles (32MHz).
isi_tx_enable	1	In	ISI transmit enable. Active high. Enables ISI transmission of long or ping packets. ACKs may still be transmitted when this bit is 0. The value of this configuration signal should not be changed while there are valid packets in the Tx buffer.
isi_rx_enable	1	In	ISI receive enable. Active high. Enables ISI packet reception. Any activity on the ISI bus will be ignored when this signal is de-asserted. This signal should only be de-asserted if the ISI block is not required for use in the design.
isi_bit_stuff_rate[3:0]	1	In	ISI bit stuffing limit. Allows the bit stuffing counter value to be programmed. Is loaded into the 4 upper bits of the 7bit wide bit stuffing counter. The lower bits are always loaded with b111, to prevent bit stuffing for less than 7 consecutive ones or zeroes. E.g. b000 : stuff_count = b0000111 : bit stuff after 7 consecutive 0/1 b111 : stuff_count = b1111111 : bit stuff after127 consecutive 0/1
Serial Link Signals			

isi_ser_data_in[3:0]	4	In	ISI Serial data inputs. Each bit corresponds to a separate serial link.
isi_ser_data_out[3:0]	4	Out	ISI Serial data outputs. Each bit corresponds to a separate serial link.
isi_ser_data_en[3:0]	4	Out	ISI Serial data driver enables. Active high. Each bit corresponds to a separate serial link.
Tx Packet Buffer			
isi_tx_wr_en	1	In	ISI Tx FIFO write enable. Active high. Asserting *isi_tx_wr_en* will write the 64 bit data on *isi_tx_wr_data* to the FIFO, providing that space is available in the FIFO. If *isi_tx_wr_en* remains asserted after the last entry in the current packet is written, the write operation will wrap around to the start of the next packet, providing that space is available for a second packet in the FIFO.
isi_tx_wr_data[63:0]	64	In	ISI Tx FIFO write data.
isi_tx_ping	1	In	ISI Tx FIFO ping packet select. Active high. Asserting *isi_tx_ping* will queue a ping packet for transmission, as opposed to a long packet. Although there is no data payload for a ping packet, a packet location in the FIFO is used as a 'place holder' for the ping packet. Any data written to the associated packet location in the FIFO will be discarded when the ping packet is transmitted.
isi_tx_id[3:0]	5	In	ISI Tx FIFO packet ID. ISI ID for each packet written to the FIFO. Registered when the last entry of the packet is written.
isi_tx_sub_id	1	In	ISI Tx FIFO packet sub ID. ISI sub ID for each packet written to the FIFO. Registered when the last entry of the packet is written.
isi_tx_pkt_count[1:0]	2	Out	ISI Tx FIFO packet count. Indicates the number of packets contained in the FIFO. The FIFO has a capacity of 2 x 256 bit packets. Range is b00->b10.
isi_tx_word_count[2:0]	3	Out	ISI Tx FIFO current packet word count. Indicates the number of words contained in the current Tx packet location of the Tx FIFO. Each packet location has a capacity of 4 x 64 bit words. Range is b000->b100.

isi_tx_empty	1	Out	ISI Tx FIFO empty. Active high. Indicates that no packets are present in the FIFO.
isi_tx_full	1	Out	ISI Tx FIFO full. Active high. Indicates that 2 packets are present in the FIFO, therefore no more packets can be transmitted.
isi_tx_over_flow	1	Out	ISI Tx FIFO over flow. Active high. Indicates that a write operation was performed on a full FIFO. The write operation will have no effect on the contents of the FIFO or the write pointer.
isi_tx_error	1	Out	ISI Tx FIFO error. Active high. Indicates that an error occurred while transmitting the packet currently at the head of the FIFO. This will happen if the number of transmission attempts exceeds *isi_tx_retries*.
isi_tx_desc[2:0]	3	Out	ISI Tx packet descriptor field. ISI packet descriptor field for the packet currently at the head of the FIFO. See Table for details. Only valid when *isi_tx_empty*=0, i.e. when there is a valid packet in the FIFO.
isi_tx_addr[4:0]	5	Out	ISI Tx packet address field. ISI address field for the packet currently at the head of the FIFO. See Table for details. Only valid when *isi_tx_empty*=0, i.e. when there is a valid packet in the FIFO.
Rx Packet FIFO			
isi_rx_rd_en	1	In	ISI Rx FIFO read enable. Active high. Asserting *isi_rx_rd_en* will drive *isi_rx_rd_data* with valid data, from the Rx packet at the head of the FIFO, providing that data is available in the FIFO. If *isi_rx_rd_en* remains asserted after the last entry is read from the current packet, the read operation will wrap around to the start of the next packet, providing that a second packet is available in the FIFO.
isi_rx_rd_data[63:0]	64	Out	ISI Rx FIFO read data.
isi_rx_sub_id	1	Out	ISI Rx packet sub ID. Indicates the ISI sub ID associated with the packet at the head of the Rx FIFO.
isi_rx_pkt_count[1:0]	2	Out	ISI Rx FIFO packet count. Indicates the number of packets contained in the FIFO.

			The FIFO has a capacity of 2 x 256 bit packets. Range is b00->b10.
isi_rx_word_count[2:0]	3	Out	ISI Rx FIFO current packet word count. Indicates the number of words contained in the Rx packet location at the head of the FIFO. Each packet location has a capacity of 4 x 64 bit words. Range is b000->b100.
isi_rx_empty	1	Out	ISI Rx FIFO empty. Active high. Indicates that no packets are present in the FIFO.
isi_rx_full	1	Out	ISI Rx FIFO full. Active high. Indicates that 2 packets are present in the FIFO, therefore no more packets can be received.
isi_rx_over_flow	1	Out	ISI Rx FIFO over flow. Active high. Indicates that a packet was addressed to the local ISI device, but the Rx FIFO was full, resulting in a NAK.
isi_rx_under_run	1	Out	ISI Rx FIFO under run. Active high. Indicates that a read operation was performed on an empty FIFO. The invalid read will return the contents of the memory location currently addressed by the FIFO read pointer and will have no effect on the read pointer.
isi_rx_frame_error	1	Out	ISI Rx framing error. Active high. Asserted by the ISI when a framing error is detected in the received packet, which can be caused by an incorrect *Start* or *Stop* field or by bit stuffing errors. The associated packet will be dropped.
isi_rx_crc_error	1	Out	ISI Rx CRC error. Active high. Asserted by the ISI when a CRC error is detected in an incoming packet. Other than dropping the errored packet ISI reception is unaffected by a CRC Error.

12.4.6.2 ISI Serial Interface Engine (isi_sie)

There are 4 instantiations of the *isi_sie* sub block in the ISI, 1 per ISI serial link. The *isi_sie* is responsible for Rx serial data sampling, Tx serial data output and bit stuffing.

Data is sampled based on a phase detection mechanism. The incoming ISI serial data stream is over sampled 5 times per ISI bit period. The phase of the incoming data is determined by detecting transitions in the ISI serial data stream, which indicates the ISI bit boundaries. An ISI bit boundary is defined as the sample phase at which a transition was detected.

The basic functional components of the isi_sie are detailed in Figure 40. These components are simply a grouping of logical functionality and do not necessarily represent hierarchy in the design.

12.4.6.2.1   SIE Edge Detection and Data I/O

The basic structure of the data I/O and edge detection mechanism is detailed in Figure 41.

NOTE: Serial data from the receiver in the pad MUST be synchronized to the *isi_pclk* domain with a 2 stage shift register external to the ISI, to reduce the risk of metastability. *ser_data_out* and *ser_data_en* should be registered externally to the ISI.

The Rx/Tx statemachine drives *ser_data_en*, *stuff_1_en* and *stuff_0_en*. The signals *stuff_1_en* and *stuff_0_en* cause a one or a zero to be driven on *ser_data_out* when they are asserted, otherwise *fifo_rd_data* is selected.

12.4.6.2.2   SIE Rx/Tx Statemachine

The Rx/Tx statemachine is responsible for the transmission of ISI Tx data and the sampling of ISI Rx data. Each ISI bit period is 5 *isi_pclk* cycles in duration.

The Tx cycle of the Rx/Tx statemachine is illustrated in Figure 42. It generates each ISI bit that is transmitted. States tx0->tx4 represent each of the 5 *isi_pclk* phases that constitute a Tx ISI bit period. *ser_data_en* controls the tristate enable for the ISI line driver in the bidirectional pad, as shown in Figure 41. *rx_tx_cycle* is asserted during both Rx and Tx states to indicate an active Rx or Tx cycle. It is primarily used to enable bit stuffing.

NOTE: All statemachine signals are assumed to be '0' unless otherwise stated.

The Tx cycle for Tx bit stuffing when the Rx/Tx statemachine inserts a '0' into the bitstream can be seen in Figure 43.

NOTE: All statemachine signals are assumed to be '0' unless otherwise stated

The Tx cycle for Tx bit stuffing when the RxTx statemachine inserts a '1' into the bitstream can be seen in Figure 44.

NOTE: All statemachine signals are assumed to be '0' unless otherwise stated

The *tx** and *stuff** states are detailed separately for clarity. They could be easily combined when coding the statemachine, however it would be better for verification and debugging if they were kept separate.

The Rx cycle of the ISI Rx/Tx statemachine is detailed in Figure 45. The Rx cycle of the Rx/Tx Statemachine, samples each ISI bit that is received. States rx0->rx4 represent each of the 5 *isi_pclk* phases that constitute a Rx ISI bit period.

The optimum sample position for an ideal ISI bit period is 2 *isi_pclk* cycles after the ISI bit boundary sample, which should result in a data sample close to the centre of the ISI bit period.

rx_sample is asserted during the *rx2* state to indicate a valid ISI data sample on *rx_bit,* unless the bit should be stripped when flagged by the bit stuffing statemachine, in which case *rx_sample* is not asserted during *rx2* and the bit is not written to the FIFO. When *edge* is asserted, it resets the Rx cycle to the rx0 state, from any rx state. This is how the *isi_sie* tracks the phase of the incoming data. The Rx cycle will cycle through states rx0->rx4 until *edge* is asserted to reset the sample phase, or a *tx_req* is asserted indicating that the ISI needs to transmit.

Due to the 5 times oversampling a maximum phase error of 0.4 of an ISI bit period (2 *isi_pclk* cycles out of 5) can be tolerated.

NOTE: All statemachine signals are assumed to be '0' unless otherwise stated.

An example of the Tx data generation mechanism is detailed in Figure 46. *tx_req* and *fifo_wr_tx* are driven by the framer block.

An example of the Rx data sampling functional timing is detailed in Figure 47. The dashed lines on the *ser_data_in_ff* signal indicate where the Rx/Tx statemachine perceived the bit boundary to be, based on the phase of the last ISI bit boundary. It can be seen that data is sampled during the same phase as the previous bit was, in the absence of a transition.

5   12.4.6.2.3   SIE Rx/Tx FIFO

The Rx/Tx FIFO is a 7 x 1 bit synchronous look-ahead FIFO that is shared for Tx and Rx operations. It is required to absorb any Rx/Tx latency caused by bit stripping/stuffing on a per ISI line basis, i.e. some ISI lines may require bit stripping/stuffing during an ISI bit period while the others may not, which would lead to a loss of synchronization between the data of the different ISI

10   lines, if a FIFO were not present in each *isi_sie*.

The basic functional components of the FIFO are detailed in Figure 48. *tx_ready* is driven by the Rx/Tx statemachine and selects which signals control the read and write operations. *tx_ready*=1 during ISI transmission and selects the *fifo_*tx* control and data signals. *tx_ready*=0 during ISI reception and selects the *fifo_*rx* control and data signals. *fifo_reset* is driven by the Rx/Tx

15   statemachine. It is active high and resets the FIFO and associated logic before/after transmitting a packet to discard any residual data.

The size of the FIFO is based on the maximum bit stuffing frequency and the size of the shift register used to segment/re-assemble the multiple serial streams in the ISI framing logic. The maximum bit stuffing frequency is every 7 consecutive ones or zeroes. The shift register used is 32

20   bits wide. This implies that the maximum number of stuffed bits encountered in the time it takes to fill/empty the shift register if 4. This would suggest that 4 x 1 bit would be the minimum *ideal* size of the FIFO. However it is necessary to allow for different skew and phase error between the ISI lines; hence a 7 x 1 bit FIFO.

The FIFO is controlled by the *isi_sie* during packet reception and is controlled by the *isi_frame* block

25   during packet transmission. This is illustrated in Figure 49. The signal *tx_ready* selects which mode the FIFO control signals operate in. When *tx_ready*=0, i.e. Rx mode, the *isi_sie* control signals *rx_sample*, *fifo_rd_rx* and *ser_data_in_ff* are selected. When *tx_ready*=1, i.e. Tx mode, the *sie_frame* control signals *fifo_wr_tx*, *fifo_rd_tx* and *fifo_wr_data_tx* are selected.

12.4.6.3 Bit Stuffing

30   Programmable bit stuffing is implemented in the *isi_sie*. This is to allow the system to determine the amount of bit stuffing necessary for a specific ISI system devices. It is unlikely that bit stuffing would be required in a system using a 100ppm rated crystal. However, a programmable bit stuffing implementation is much more versatile and robust.

. The bit stuffing logic consists of a counter and a statemachine that track the number of consecutive

35   ones or zeroes that are transmitted or received and flags the Rx/Tx statemachine when the bit stuffing limit has been reached. The counter, *stuff_count*, is a 7 bit counter, which decrements when *rx_sample* is asserted on a Rx cycle or when *fifo_rd_tx* is asserted on a Tx cycle. The upper 4 bits of *stuff_count* are loaded with *isi_bit_stuff_rate*. The lower 3 bits of *stuff_count* are always loaded with b111, i.e. for *isi_bit_stuff_rate* = b000, the counter would be loaded with b0000111. This is to

prevent bit stuffing for less than 7 consecutive ones or zeroes. This allows the bit stuffing limit to be set in the range 7->127 consecutive ones or zeroes.

NOTE: It is extremely important that a change in the bit stuffing rate, *isi_bit_stuff_rate*, is carefully co-ordinated between ISI devices in a system. It is obvious that ISI devices will not be able to communicate reliably with each other with different bit stuffing settings. It is recommended that all ISI devices in a system default to the safest bit stuffing rate (*isi_bit_stuff_rate* = b000) at reset. The system can then co-ordinate the change to an optimum bit stuffing rate.

The ISI bit stuffing statemachine Tx cycle is shown in Figure 50. The counter is loaded when *stuff_count_load* is asserted.

NOTE: All statemachine signals are assumed to be '0' unless otherwise stated.

The ISI bit stuffing statemachine Rx cycle is shown in Figure 51. It should be noted that the statemachine enters the strip state when *stuff_count=0x2*. This is because the statemachine can only transition to *rx0* or *rx1* when *rx_sample* is asserted as it needs to be synchronized to changes in sampling phase introduced by the Rx/Tx statemachine. Therefore a one or a zero has already been sampled by the time it enters *rx0* or *rx1*. This is not the case for the Tx cycle, as it will always have a stable 5 isi_pclk cycles per bit period and relies purely on the data value when entering *tx0* or *tx1*. The Tx cycle therefore enters *stuff1* or *stuff0* when *stuff_count=0x1*.

NOTE: All statemachine signals are assumed to be '0' unless otherwise stated.

12.4.6.4   ISI Framing and CRC sub-block (isi_frame)

12.4.6.4.1   CRC Generation/Checking

A Cyclic Redundancy Checksum (CRC) is calculated over all fields except the start and stop fields for each long or ping packet transmitted. The receiving ISI device will perform the same calculation on the received packet to verify the integrity of the packet. The procedure used in the CRC generation/checking is the same as the Frame Checking Sequence (FCS) procedure used in HDLC, detailed in ITU-T Recommendation T30[39].

For generation/checking of the CRC field, the shift register illustrated in Figure 52 is used to perform the modulo 2 division on the packet contents by the polynomial $G(x) = x^{16} + x^{12} + x^5 + 1$.

To generate the CRC for a transmitted packet, where T(x) = [Packet Descriptor field, Address field, Data Payload field] (a ping packet will not contain a data payload field).

- Set the shift register to 0xFFFF.
- Shift *T(x)* through the shift register, LSB first. This can occur in parallel with the packet transmission.
- Once the each bit of *T(x)* has been shifted through the register, it will contain the remainder of the modulo 2 division *T(x)*/G(x).
- Perform a ones complement of the register contents, giving the CRC field which is transmitted MSB first, immediately following the last bit of *M(x*

  To check the CRC for a received packet, where R(x) = [Packet Descriptor field, Address field, Data Payload field, CRC field] (a ping packet will not contain a data payload field).

- Set the shift register to 0xFFFF.

- Shift R(x) through the shift register, LSB first. This can occur in parallel with the packet reception.
- Once each bit of the packet has been shifted through the register, it will contain the remainder of the modulo 2 division $R(x)/G(x)$.
- The remainder should equal b0001110100001111, for a packet without errors.

12.5 CTRL (CONTROL SUB-BLOCK)

12.5.1 Overview

The CTRL is responsible for high level control of the SCB sub-blocks and coordinating access between them. All control and status registers for the SCB are contained within the CTRL and are accessed via the CPU interface. The other major components of the CTRL are the SCB Map logic and the DMA Manager logic.

12.5.2 SCB Mapping

In order to support maximum flexibility when moving data through a multi-SoPEC system it is possible to map any USB endpoint onto either DMAChannel within any SoPEC in the system. The SCB map, and indeed the SCB itself is based around the concept of an ISIId and an ISISubId. Each SoPEC in the system has a unique ISIId and two ISISubIds, namely ISISubId0 and ISISubId1. We use the convention that ISISubId0 corresponds to DMAChannel0 in each SoPEC and ISISubId1 corresponds to DMAChannel1. The naming convention for the ISIId is shown in Table 35 below and this would correspond to a multi-SoPEC system such as that shown in Figure 27. We use the term ISIId instead of SoPECId to avoid confusion with the unique ChipID used to create the SoPEC_id and SoPEC_id_key (see chapter 17 and [9] for more details).

Table 35. ISIId naming convention

ISIId	SoPEC to which it refers
0 - 14	Standard device ISIIds (0 is the power-on reset value)
15	Broadcast ISIId

The combined ISIId and ISISubId therefore allows the ISI to address DMAChannel0 or DMAChannel1 on any SoPEC device in the system. The ISI, DMA manager and SCB map hardware use the ISIId and ISISubId to handle the different data streams that are active in a multi-SoPEC system as does the software running on the CPU of each SoPEC. In this document we will identify DMAChannels as *ISIx.y* where *x* is the ISIId and *y* is the ISISubId. Thus ISI2.1 refers to DMAChannel1 of ISISlave2. Any data sent to a broadcast channel, i.e. ISI15.0 or ISI15.1, are received by every ISI device in the system including the ISIMaster (which may be an ISI-Bridge). The USB device controller and software stacks however have no understanding of the ISIId and ISISubId but the Silverbrook printer driver software running on the external host does make use of the ISIId and ISISubId. USB is simply used as a data transport - the mapping of USB device endpoints onto ISIId and SubId is communicated from the external host Silverbrook code to the SoPEC Silverbrook code through USB control (or possibly bulk data) messages i.e. the mapping

information is simply data payload as far as USB is concerned. The code running on SoPEC is responsible for parsing these messages and configuring the SCB accordingly.

The use of just two DMAChannels places some limitations on what can be achieved without software intervention. For every SoPEC in the system there are more potential sources of data than there are sinks. For example an ISISlave could receive both control and data messages from the ISIMaster SoPEC in addition to control and data from the external host, either specifically addressed to that particular ISISlave or over the broadcast ISI channel. However all ISISlaves only have two possible data sinks, i.e. DMAChannel0 and DMAChannel1. Another example is the ISIMaster in a multi-SoPEC system which may receive control messages from each SoPEC in addition to control and data information from the external host (e.g. over USB). In this case all of the control messages are in contention for access to DMAChannel0. We resolve these potential conflicts by adopting the following conventions:

1) Control messages may be interleaved in a memory buffer: The memory buffer that the DMAChannel0 points to should be regarded as a central pool of control messages. Every control message must contain fields that identify the size of the message, the source and the destination of the control message. Control messages may therefore be multiplexed over a DMAChannel which allows several control message sources to address the same DMAChannel. Furthermore, if SoPEC-type control messages contain source and destination fields it is possible for the external host to send control messages to individual SoPECs over the ISI15.0 broadcast channel.

2) Data messages should not be interleaved in a memory buffer: As data messages are typically part of a much larger block of data that is being transferred it is not possible to control their contents in the same manner as is possible with the control messages. Furthermore we do not want the CPU to have to perform reassembly of data blocks. Data messages from different sources cannot be interleaved over the same DMAChannel - the SCB map must be reconfigured each time a different data source is given access to the DMAChannel.

3) Every reconfiguration of the SCB map requires the exchange of control messages: SoPEC's SCB map reset state is shown in Table   and any subsequent modifications to this map require the exchange of control messages between the SoPEC and the external host. As the external host is expected to control the movement of data in any SoPEC system it is anticipated that all changes to the SCB map will be performed in response to a request from the external host. While the SoPEC could autonomously reconfigure the SCB map (this is entirely up to the software running on the SoPEC) it should not do so without informing the external host in order to avoid data being mis-routed.

An example of the above conventions in operation is worked through in section 12.5.2.3.

12.5.2.1 SCB map rules

The operation of the SCB map is described by these 2 rules:

Rule 1: A packet is routed to the DMA manager if it originates from the USB device core and has an ISIId that matches the local SoPEC ISIId.

Rule 2: A packet is routed to the ISI if it originates from the CPU or has an ISIId that does not match the local SoPEC ISIId.

137

If the CPU erroneously addresses a packet to the ISIId contained in the *ISIId* register (i.e. the ISIId of the local SoPEC) then that packet will be transmitted on the ISI rather than be sent to the DMA manager. While this will usually cause an error on the ISI there is one situation where it could be beneficial, namely for initial dialog in a 2 SoPEC system as both devices come out of reset with an

5 ISIId of 0.

12.5.2.2 External host to ISIMaster SoPEC communication

Although the SCB map configuration is independent of ISIMaster status, the following discussion on SCB map configurations assumes the ISIMaster is a SoPEC device rather than an ISI bridge chip, and that only a single USB connection to the external host is present. The information should apply

10 broadly to an ISI-Bridge but we focus here on an ISIMaster SoPEC for clarity.

As the ISIMaster SoPEC represents the printer device on the PC USB bus it is required by the USB specification to have a dedicated control endpoint, EP0. At boot time the ISIMaster SoPEC will also require a bulk data endpoint to facilitate the transfer of program code from the external host. The simplest SCB map configuration, i.e. for a single stand-alone SoPEC, is sufficient for external host

15 to ISIMaster SoPEC communication and is shown in Table 36.

Table 36. Single SoPEC SCB map configuration

Source	Sink
EP0	ISI0.0
EP1	ISI0.1
EP2	nc
EP3	nc
EP4	nc

In this configuration all USB control information exchanged between the external host and SoPEC over EP0 (which is the only bidirectional USB endpoint). SoPEC specific control information (printer

20 status, DNC info etc.) is also exchanged over EP0.

All packets sent to the external host from SoPEC over EP0 must be written into the DMA mapped EP buffer by the CPU (LEON-PC dataflow in Figure 29). All packets sent from the external host to SoPEC are placed in DRAM by the DMA Manager, where they can be read by the CPU (PC-DIU dataflow in Figure 29). This asymmetry is because in a multi-SoPEC environment the CPU will need

25 to examine all incoming control messages (i.e. messages that have arrived over DMAChannel0) to ascertain their source and destination (i.e. they could be from an ISISlave and destined for the external host) and so the additional overhead in having the CPU move the short control messages to the EP0 FIFO is relatively small. Furthermore we wish to avoid making the SCB more complicated than necessary, particularly when there is no significant performance gain to be had as

30 the control traffic will be relatively low bandwidth.

The above mechanisms are appropriate for the types of communication outlined in sections 12.1.2.1.1 through 12.1.2.1.4

12.5.2.3 Broadcast communication

138

The SCB configuration for broadcast communication is also the default, post power-on reset, configuration for SoPEC and is shown in Table 37.

5   Table 37. Default SoPEC SCB map configuration

Source	Sink
EP0	ISI0.0
EP1	ISI0.1
EP2	ISI15.0
EP3	ISI15.1
EP4	ISI1.1

USB endpoints EP2 and EP3 are mapped onto ISISubID0 and ISISubId1 of ISIId15 (the broadcast ISIId channel). EP0 is used for control messages as before and EP1 is a bulk data endpoint for the
10   ISIMaster SoPEC. Depending on what is convenient for the boot loader software, EP1 may or may not be used during the initial program download, but EP1 is highly likely to be used for compressed page or other program downloads later. For this reason it is part of the default configuration. In this setup the USB device configuration will take place, as it always must, by exchanging messages over the control channel (EP0).
15   One possible boot mechanism is where the external host sends the bootloader1 program code to all SoPECs by broadcasting it over EP3. Each SoPEC in the system then authenticates and executes the bootloader1 program. The ISIMaster SoPEC then polls each ISISlave (over the ISIx.0 channel). Each ISISlave ascertains its ISIId by sampling the particular GPIO pins required by the bootloader1 and reporting its presence and status back to the ISIMaster. The ISIMaster then passes this
20   information back to the external host over EP0. Thus both the external host and the ISIMaster have knowledge of the number of SoPECs, and their ISIIds, in the system. The external host may then reconfigure the SCB map to better optimise the SCB resources for the particular multi-SoPEC system. This could involve simplifying the default configuration to a single SoPEC system or remapping the broadcast channels onto DMAChannels in individual ISISlaves.
25   The following steps are required to reconfigure the SCB map from the configuration depicted in Table   to one where EP3 is mapped onto ISI1.0:

1) The external host sends a control message(s) to the ISIMaster SoPEC requesting that USB EP3 be remapped to ISI1.0

2) The ISIMaster SoPEC sends a control message to the external host informing it that EP3 has
30   now been mapped to ISI1.0 (and therefore the external host knows that the previous mapping of ISI15.1 is no longer available through EP3).

3) The external host may now send control messages directly to ISISlave1 without requiring any CPU intervention on the ISIMaster SoPEC

12.5.2.4 External host to ISISlave SoPEC communication

139

If the ISIMaster is configured correctly (e.g. when the ISIMaster is a SoPEC, and that SoPEC's SCB map is configured correctly) then data sent from the external host destined for an ISISlave will be transmitted on the ISI with the correct address. The ISI automatically forwards any data addressed to it (including broadcast data) to the DMA channel with the appropriate ISISubId. If the ISISlave has data to send to the external host it must do so by sending a control message to the ISIMaster identifying the external host as the intended recipient. It is then the ISIMaster's responsibility to forward this message to the external host.

With this configuration the external host can communicate with the ISISlave via broadcast messages only and this is the mechanism by which the bootloader1 program is downloaded. The ISISlave is unable to communicate with the external host (or the ISIMaster) until the bootlloader1 program has successfully executed and the ISISlave has determined what its ISIId is. After the bootloader1 program (and possibly other programs) has executed the SCB map of the ISIMaster may be reconfigured to reflect the most appropriate topology for the particular multi-SoPEC system it is part of.

All communication from an ISISlave to external host is either achieved directly (if there is a direct USB connection present for example) or by sending messages via the ISIMaster. The ISISlave can never initiate communication to the external host. If an ISISlave wishes to send a message to the external host via the ISIMaster it must wait until it is pinged by the ISIMaster and then send a the message in a long packet addressed to the ISIMaster. When the ISIMaster receives the message from the ISISlave it first examines it to determine the intended destination and will then copy it into the EP0 FIFO for transmission to the external host. The software running on the ISIMaster is responsible for any arbitration between messages from different sources (including itself) that are all destined for the external host.

The above mechanisms are appropriate for the types of communication outlined in sections 12.1.2.1.5 and 12.1.2.1.6.

12.5.2.5 ISIMaster to ISISlave communication

All ISIMaster to ISISlave communication takes place over the ISI. Immediately after reset this can only be by means of broadcast messages. Once the bootloader1 program has successfully executed on all SoPECs in a multi-SoPEC system the ISIMaster can communicate with each SoPEC on an individual basis.

If an ISISlave wishes to send a message to the ISIMaster it may do so in response to a ping packet from the ISIMaster. When the ISIMaster receives the message from the ISISlave it must interpret the message to determine if the message contains information required to be sent to the external host. In the case of the ISIMaster being a SoPEC, software will transfer the appropriate information into the EP0 FIFO for transmission to the external host.

The above mechanisms are appropriate for the types of communication outlined in sections 12.1.2.3.3 and 12.1.2.3.4.

12.5.2.6 ISISlave to ISISlave communication

ISISlave to ISISlave communication is expected to be limited to two special cases: (a) when the PrintMaster is not the ISIMaster and (b) when a storage SoPEC is used. When the PrintMaster is

not the ISIMaster then it will need to send control messages (and receive responses to these messages) to other ISISlaves. When a storage SoPEC is present it may need to send data to each SoPEC in the system. All ISISlave to ISISlave communication will take place in response to ping messages from the ISIMaster.

5   *12.5.2.7   Use of the SCB map in an ISISlave with a external host connection*

After reset any SoPEC (regardless of ISIMaster/Slave status) with an active USB connection will route packets from EP0,1 to DMA channels 0,1 because the default SCB map is to map EP0 to ISIId0.0 and EP1 to ISIId0.1 and the default ISIId is 0. At some later time the SoPEC learns its true ISIId for the system it is in and re-configures its ISIId and SCB map registers accordingly. Thus if

10   the true ISIId is 3 the external host could reconfigure the SCB map so that EP0 and EP1 (or any other endpoints for that matter) map to ISIId3.0 and 3.1 respectively. The co-ordination of the updating of the ISIId registers and the SCB map is a matter for software to take care of. While the *AutoMasterEnable* bit of the *ISICntrl* register is set the external host must not send packets down EP2-4 of the USB connection to the device intended to be an ISISlave. When *AutoMasterEnable*

15   has been cleared the external host may send data down any endpoint of the USB connection to the ISISlave.

The SCB map of an ISISlave can be configured to route packets from any EP to any ISIId.ISISubId (just as an ISIMaster can). As with an ISIMaster these packets will end up in the SCBTxBuffer but while an ISIMaster would just transmit them when it got a local access slot (from ping arbitration)

20   the ISISlave can only transmit them in response to a ping. All this would happen without CPU intervention on the ISISlave (or ISIMaster) and as long as the ping frequency is sufficiently high it would enable maximum use of the bandwidth on both USB buses.

12.5.3   DMA Manager

The DMA manager manages the flow of data between the SCB and the embedded DRAM. Whilst

25   the CPU could be used for the movement of data in SoPEC, a DMA manager is a more efficient solution as it will handle data in a more predictable fashion with less latency and requiring less buffering. Furthermore a DMA manager is required to support the ISI transfer speed and to ensure that the SoPEC could be used with a high speed ISI-Bridge chip in the future.

The DMA manager utilizes 2 write channels (DMAChannel0, DMAChannel1) and 1 read/write

30   channel (DMAChannel2) to provide 2 independent modes of access to DRAM via the DIU interface:

•   USBD/ISI type access.

•   USBH type access.

DIU read and write access is in bursts of 4x64 bit words. Byte aligned write enables are provided for write access. Data for DIU write accesses will be read directly from the buffers contained in the

35   respective SCB sub-blocks. There is no internal SCB DMA buffer. The DMA manager handles all issues relating to byte/ word/longword address alignment, data endianness and transaction scheduling. If a DMA channel is disabled during a DMA access, the access will be completed. Arbitration will be performed between the following DIU access requests:

•   USBD write request.

40   •   ISI write request.

- USBH write request.
- USBH read request.

DMAChannel0 will have absolute priority over any DMA requestors. In the absence of DMAChannel0 DMA requests, arbitration will be performed in a round robin manner, on a per cycle basis over the other channels.

12.5.3.1 DMA Effective Bandwidth

The DIU bandwidth available to the DMA manager must be set to ensure adequate bandwidth for all data sources, to avoid back pressure on the USB and the ISI. This is achieved by setting the output (i.e. DIU) bandwidth to be greater than the combined input bandwidths (i.e. USBD + USBH + ISI). The required bandwidth is expected to be 160 Mbits/s (1 bit/cycle @ 160MHz). The guaranteed DIU bandwidth for the SCB is programmable and may need further analysis once there is better knowledge of the data throughput from the USB IP cores.

12.5.3.2 USBD/ISI DMA access

The DMA manager uses the two independent unidirectional write channels for this type of DMA access, one for each ISISubId, to control the movement of data. Both DMAChannel0 and DMAChannel1 only support write operation and can transfer data from any USB device DMA mapped EP buffer and from the ISI receive buffer to separate circular buffers in DRAM, corresponding to each DMA channel.

While the DMA manager performs the work of moving data the CPU controls the destination and relative timing of data flows to and from the DRAM. The management of the DRAM data buffers requires the CPU to have accurate and timely visibility of both the DMA and PEP memory usage. In other words when the PEP has completed processing of a page band the CPU needs to be aware of the fact that an area of memory has been freed up to receive incoming data. The management of these buffers may also be performed by the external host.

12.5.3.2.1 Circular buffer operation

The DMA manager supports the use of circular buffers for both DMAChannels. Each circular buffer is controlled by 5 registers: *DMAnBottomAdr, DMAnTopAdr, DMAnMaxAdr, DMAnCurrWPtr* and *DMAnIntAdr*. The operation of the circular buffers is shown in Figure 53 below.

Here we see two snapshots of the status of a circular buffer with (b) occurring sometime after (a) and some CPU writes to the registers occurring in between (a) and (b). These CPU writes are most likely to be as a result of a finished band interrupt (which frees up buffer space) but could also have occurred in a DMA interrupt service routine resulting from *DMAnIntAdr* being hit. The DMA manager will continue filling the free buffer space depicted in (a), advancing the *DMAnCurrWPtr* after each write to the DIU. Note that the *DMACurrWPtr* register always points to the next address the DMA manager will write to. When the DMA manager reaches the address in *DMAnIntAdr* (i.e. *DMACurrWPtr = DMAnIntAdr*) it will generate an interrupt if the *DMAnIntAdrMask* bit in the *DMAMask* register is set. The purpose of the *DMAnIntAdr* register is to alert the CPU that data (such as a control message or a page or band header) has arrived that it needs to process. The interrupt routine servicing the DMA interrupt will change the *DMAnIntAdr* value to the next location that data of interest to the CPU will have arrived by.

In the scenario shown in Figure 53 the CPU has determined (most likely as a result of a finished band interrupt) that the filled buffer space in (a) has been freed up and is therefore available to receive more data. The CPU therefore moves the *DMAnMaxAdr* to the end of the section that has been freed up and moves the *DMAnIntAdr* address to an appropriate offset from the *DMAnMaxAdr*

5   address. The DMA manager continues to fill the free buffer space and when it reaches the address in *DMAnTopAdr* it wraps around to the address in *DMAnBottomAdr* and continues from there. DMA transfers will continue indefinitely in this fashion until the DMA manager reaches the address in the *DMAnMaxAdr* register.

The circular buffer is initialized by writing the top and bottom addresses to the *DMAnTopAdr* and

10   *DMAnBottomAdr* registers, writing the start address (which does not have to be the same as the *DMAnBottomAdr* even though it usually will be) to the *DMAnCurrWPtr* register and appropriate addresses to the *DMAnIntAdr* and *DMAnMaxAdr* registers. The DMA operation will not commence until a 1 has been written to the relevant bit of the *DMAChanEn* register.

While it is possible to modify the *DMAnTopAdr* and *DMAnBottomAdr* registers after the DMA has

15   started it should be done with caution. The *DMAnCurrWPtr* register should not be written to while the DMAChannel is in operation. DMA operation may be stalled at any time by clearing the appropriate bit of the *DMAChanEn* register or by disabling an SCB mapping or ISI receive operation.

12.5.3.2.2   Non-standard buffer operation

20   The DMA manager was designed primarily for use with a circular buffer. However because the DMA pointers are tested for equality (i.e. interrupts generated when *DMAnCurrWPtr = DMAIntAdr* or *DMAnCurrWPtr = DMAMaxAdr*) and no bounds checking is performed on their values (i.e. neither *DMAnIntAdr* nor *DMAnMaxAdr* are checked to see if they lie between *DMAnBottomAdr* and *DMAnTopAdr*) a number of non-standard buffer arrangements are possible. These include:

25   • Dustbin buffer: If *DMAnBottomAdr, DMAnTopAdr* and *DMAnCurrWPtr* all point to the same location and both *DMAnIntAdr* and *DMAnMaxAdr* point to anywhere else then all data for that DMA channel will be dumped into the same location without ever generating an interrupt. This is the equivalent to writing to /dev/null on Unix systems.

       • Linear buffer: If *DMAnMaxAdr* and *DMAnTopAdr* have the same value then the DMA

30   manager will simply fill from *DMAnBottomAdr* to *DMAnTopAdr* and then stop. *DMAnIntAdr* should be outside this buffer or have its interrupt disabled.

12.5.3.3 USBH DMA access

The USBH requires DMA access to DRAM in to provide a communication channel between the USB HC and the USB HCD via a shared memory resource. The DMA manager uses two

35   independent channels for this type of DMA access, one for reads and one for writes. The DRAM addresses provided to the DIU interface are generated based on addresses defined in the USB HC core operational registers, in USBH section 12.3.

12.5.3.4 Cache coherency

As the CPU will be processing some of the data transferred (particularly control messages and

40   page/band headers) into DRAM by the DMA manager, care needs to be taken to ensure that the

data it uses is the most recently transferred data. Because the DMA manager will be updating the circular buffers in DRAM without the knowledge of the cache controller logic in the LEON CPU core the contents of the cache can become outdated. This situation can be easily handled by software, for example by flushing the relevant cache lines, and so there is no hardware support to enforce

5   cache coherency.

12.5.4   ISI transmit buffer arbitration

The SCB control logic will arbitrate access to the ISI transmit buffer (ISITxBuffer) interface on the ISI block. There are two sources of ISI Tx packets:

- CPUISITxBuffer, contained in the SCB control block.

10 - ISI mapped USB EP OUT buffers, contained in the USB device block.

This arbitration is controlled by the *ISITxBuffArb* register which contains a high priority bit for both the CPU and the USB. If only one of these bits is set then the corresponding source always has priority. Note that if the CPU is given absolute priority over the USB, then the software filling the ISI transmit buffer needs to ensure that sufficient USB traffic is allowed through. If both bits of the

15 *ISITxBufferArb* have the same value then arbitration will take place on a round robin basis.

The control logic will use the *USBEPnDest registers*, as it will use the *CPUISITxBuffCntrl* register, to determine the destination of the packets in these buffers. When the ISITxBuffer has space for a packet, the SCB control logic will immediately seek to refill it. Data will be transferred directly from the CPUISITxBuffer and the ISI mapped USB EP OUT buffers to the ISITxBuffer without any

20 intermediate buffering.

As the speed at which the ISITxBuffer can be emptied is at least 5 times greater than it can be filled by USB traffic, the ISI mapped USB EP OUT buffers should not overflow using the above scheme in normal operation. There are a number of scenarios which could lead to the USB EPs being temporarily blocked such as the CPU having priority, retransmissions on the ISI bus, channels

25 being enabled (*ChannelEn* bit of the *USBEPnDest* register) with data already in their associated endpoint buffers or short packets being sent on the USB. Care should be taken to ensure that the USB bandwidth is efficiently utilised at all times.

12.5.5   Implementation

12.5.5.1 CTRL Sub-block Partition

30 * Block Diagram

* Definition of I/Os

12.5.5.2 SCB Configuration Registers

The SCB register map is listed in Table 38. Registers are grouped according to which SCB sub-block their functionality is associated. All configuration registers reside in the CTRL sub-block. The

35 Reset values in the table indicates the 32 bit hex value that will be returned when the CPU reads the associated address location after reset. All Registers pre-fixed with *Hc* refer to Host Controller Operational Registers, as defined in the OHCI Spec[19].

The SCB will only allow supervisor mode accesses to data space (i.e. *cpu_acode*[1:0] = b11). All other accesses will result in *scb_cpu_berr* being asserted.

TDB: Is read access necessary for ISI Rx/Tx buffers? Could implement the ISI interface as simple FIFOs as opposed to a memory interface.

Table 38. SCB control block configuration registers

Addre ss Offset from SCB_ base	Register	#Bits	Reset	Description
CTRL				
0x000	SCBResetN	4	0x0000000F	SCB software reset. Allows individual sub-blocks to be reset separately or together. Once a reset for a block has been initiated, by writing a 0 to the relevant register field, it can not be suppressed. Each field will be set after reset. Writing 0x0 to the SCBReset register will have the same effect as CPR generated hardware reset.
0x004	SCBGo	2	0x00000000	SCB Go. Allows the ISI and CTRL sub-blocks to be selected separately or together. When go is de-asserted for a particular sub-block, its statemachines are reset to their idle states and its interface signals are de-asserted. The sub-block counters and configuration registers retain their values. When go is asserted for a particular sub-block, its counters are reset. The sub-block configuration registers retain their values, i.e. they don't get reset. The sub-block statemachines and interface signals will return to their normal mode of operation. The CTRL field should be de-asserted before disabling the clock from any part of the SCB to avoid erroneous SCB DMA requests when the clock is enabled again. NOTE: This functionality has not been provided for the USBH and USBD sub-

				blocks because of the USB IP cores that they contain. We do not have direct control over the IP core statemachines and counters, and it would cause unpredictable behaviour if the cores were disabled in this way during operation.
0x008	SCBWakeupEn	2	0x00000000	USB/ISI WakeUpEnable register
0x00C	SCBISITxBufferAr b	2	0x00000000	ISI transmit buffer access priority register.
0x010	SCBDebugSel[11: 2]	10	0x00000000	SCB Debug select register.
0x014	USBEP0Dest	7	0x00000020	This register determines which of the data sinks the data arriving in EP0 should be routed to.
0x018	USBEP1Dest	7	0x00000021	Data sink mapping for USB EP1
0x01C	USBEP2Dest	7	0x0000003E	Data sink mapping for USB EP2
0x020	USBEP3Dest	7	0x0000003F	Data sink mapping for USB EP3
0x024	USBEP4Dest	7	0x00000023	Data sink mapping for USB EP4
0x028	DMA0BottomAdr[2 1:5]	17		DMAChannel0 bottom address register.
0x02C	DMA0TopAdr[21:5]	17		DMAChannel0 top address register.
0x030	DMA0CurrWPtr[21 :5]	17		DMAChannel0 current write pointer.
0x034	DMA0IntAdr[21:5]	17		DMAChannel0 interrupt address register.
0x038	DMA0MaxAdr[21: 5]	17		DMAChannel0 max address register.
0x03C	DMA1BottomAdr[2 1:5]	17		As per *DMA0BottomAdr*.
0x040	DMA1TopAdr[21:5]	17		As per *DMA0TopAdr*.
0x044	DMA1CurrWPtr[21 :5]	17		As per *DMA0CurrWPtr*.
0x048	DMA1IntAdr[21:5]	17		As per *DMA0IntAdr*.
0x04C	DMA1MaxAdr[21: 5]	17		As per *DMA0MaxAdr*.
0x050	DMAAccessEn	3	0x00000003	DMA access enable.

0x054	DMAStatus	4	0x00000000	DMA status register.
0x058	DMAMask	4	0x00000000	DMA mask register.
0x05C - 0x098	CPUISITxBuff[7:0]	32x8	n/a	CPU ISI transmit buffer. 32-byte packet buffer, containing the payload of a CPU sourced packet destined for transmission over the ISI. The CPU has full write access to the CPUISITxBuff. NOTE: The CPU does not have read access to CPUISITxBuff. This is because the CPU is the source of the data and to avoid arbitrating read access between the CPU and the CTRL sub-block. Any CPU reads from this address space will return 0x00000000.
0x09C	CPUISITxBuffCtrl	9	0x00000000	CPU ISI transmit buffer control register.
USBD				
0x100	USBDIntStatus	19	0x00000000	USBD Interrupt event status register.
0x104	USBDISIFIFOStatus	16	0x00000000	USBD ISI mapped OUT EP packet FIFO status register.
0x108	USBDDMA0FIFO Status	8	0x00000000	USBD DMAChannel0 mapped OUT EP packet FIFO status register.
0x10C	USBDDMA1FIFO Status	8	0x00000000	USBD DMAChannel1 mapped OUT EP packet FIFO status register.
0x110	USBDResume	1	0x00000000	USBD core resume register.
0x114	USBDSetup	4	0x00000000	USBD setup/configuration register.
0x118 - 0x154	USBDEp0InBuff[15:0]	32x16	n/a	USBD EP0-IN buffer. 64-byte packet buffer in the, containing the payload of a USB packet destined for EP0-IN. The CPU has full write access to the USBDEp0InBuff. NOTE: The CPU does not have read access to USBDEp0InBuff. This is because the CPU is the source of the data and to avoid arbitrating read access between the CPU and the USB device core. Any CPU reads from this

				address space will return 0x00000000.
0x158	USBDEp0InBuffCtrl	1	0x00000000	USBD EP0-IN buffer control register.
0x15C - 0x198	USBDEp5InBuff[15:0]	32x16	n/a	USBD EP5-IN buffer. As per *USBDEp0InBuff.*
0x19C	USBDEp5InBuffCtrl	1	0x00000000	USBD EP5-IN buffer control register.
0x1A0	USBDMask	19	0x00000000	USBD interrupt mask register.
0x1A4	USBDDebug	30	0x00000000	USBD debug register.
USBH				
0x200	HcRevision			Refer to [19] for #Bits, Reset, Description.
0x204	HcControl			Refer to [19] for #Bits, Reset, Description.
0x208	HcCommandStatus			Refer to [19] for #Bits, Reset, Description.
0x20C	HcInterruptStatus			Refer to [19] for #Bits, Reset, Description.
0x210	HcInterruptEnable			Refer to [19] for #Bits, Reset, Description.
0x214	HcInterruptDisable			Refer to [19] for #Bits, Reset, Description.
0x218	HcHCCA			Refer to [19] for #Bits, Reset, Description.
0x21C	HcPeriodCurrentED			Refer to [19] for #Bits, Reset, Description.
0x220	HcControlHeadED			Refer to [19] for #Bits, Reset, Description.
0x224	HcControlCurrentED			Refer to [19] for #Bits, Reset, Description.
0x228	HcBulkHeadED			Refer to [19] for #Bits, Reset, Description.
0x22C	HcBulkCurrentED			Refer to [19] for #Bits, Reset, Description.
0x230	HcDoneHead			Refer to [19] for #Bits, Reset, Description.
0x234	HcFmInterval			Refer to [19] for #Bits, Reset, Description.
0x238	HcFmRemaining			Refer to [19] for #Bits, Reset,

				Description.
0x23C	HcFmNumber			Refer to [19] for #Bits, Reset, Description.
0x240	HcPeriodicStart			Refer to [19] for #Bits, Reset, Description.
0x244	HcLSTheshold			Refer to [19] for #Bits, Reset, Description.
0x248	HcRhDescriptorA			Refer to [19] for #Bits, Reset, Description.
0x24C	HcRhDescriptorB			Refer to [19] for #Bits, Reset, Description.
0x250	HcRhStatus			Refer to [19] for #Bits, Reset, Description.
0x254	HcRhPortStatus[1]			Refer to [19] for #Bits, Reset, Description.
0x258	USBHStatus	3	0x00000000	USBH status register.
0x25C	USBHMask	2	0x00000000	USBH interrupt mask register.
0x260	USBHDebug	2	0x00000000	USBH debug register.
ISI				
0x300	ISICntrl	4	0x0000000B	ISI Control register
0x304	ISIId	4	0x00000000	ISIId for this SoPEC.
0x308	ISINumRetries	4	0x00000002	Number of ISI retransmissions register.
0x30C	ISIPingSchedule0	15	0x00000000	ISI Ping schedule 0 register.
0x310	ISIPingSchedule1	15	0x00000000	ISI Ping schedule 1 register.
0x314	ISIPingSchedule2	15	0x00000000	ISI Ping schedule 2 register.
0x318	ISITotalPeriod	4	0x0000000F	Reload value of the *ISITotalPeriod* counter.
0x31C	ISILocalPeriod	4	0x0000000F	Reload value of the *ISILocalPeriod* counter.
0x320	ISIIntStatus	4	0x00000000	ISI interrupt status register.
0x324	ISITxBuffStatus	27	0x00000000	ISI Tx buffer status register.
0x328	ISIRxBuffStatus	27	0x00000000	ISI Rx buffer status register.
0x32C	ISIMask	4	0x00000000	ISI Interrupt mask register.
0x330 - 0x34C	ISITxBuffEntry0[7:0]	32x8	n/a	ISI transmit Buff, packet entry #0. 32-byte packet entry in the *ISITxBuff*, containing the payload of an ISI Tx packet. CPU read access to *ISITxBuffEntry0* is provided for observability only i.e. CPU

				reads of the *ISITxBuffEntry0* do not alter the state of the buffer. The CPU does not have write access to the *ISITxBuffEntry0*.
0x350 - 0x36C	ISITxBuffEntry1[7: 0]	32x8	n/a	ISI transmit Buff, packet entry #1. As per *ISITxBuffEntry0*.
0x370 - 0x38C	ISIRxBuffEntry0[7: 0]	32x8	n/a	ISI receive Buff, packet entry #0. 32-byte packet entry in the *ISIRxBuff*, containing the payload of an ISI Rx packet. Note that the only error-free long packets are placed in the *ISIRxBuffEntry0*. Both ping and ACKs are consumed in the ISI. CPU access to *ISIRxBuffEntry0* is provided for observability only i.e. CPU reads of the *ISIRxBuffEntry0* do not alter the state of the buffer.
0x390 - 0x3AC	ISIRxBuffEntry1[7: 0]	32x8	n/a	ISI receive Buff, packet entry #1. As per *ISIRxBuffEntry0*.
0x3B0	ISISubId0Seq	1	0x00000000	ISI sub ID 0 sequence bit register.
0x3B4	ISISubId1Seq	1	0x00000000	ISI sub ID 1 sequence bit register.
0x3B8	ISISubIdSeqMask	2	0x00000000	ISI sub ID sequence bit mask register.
0x3BC	ISINumPins	1	0x00000000	ISI number of pins register.
0x3C0	ISITurnAround	4	0x0000000F	ISI bus turn around register.
0x3C4	ISITShortReplyWi n	5	0x0000001F	ISI short packet reply window.
0x3C8	ISITLongReplyWin	9	0x000001FF	ISI long packet reply window.
0x3CC	ISIDebug	4	0x00000000	ISI debug register.

A detailed description of each register format follows. The CPU has full read access to all registers. Write access to the fields of each register is defined as:

- Full: The CPU has full write access to the field, i.e. the CPU can write a 1 or a 0 to each bit.

- Clear: The CPU can clear the field by writing a 1 to each bit. Writing a 0 to this type of field will have no effect.

- None: The CPU has no write access to the field, i.e. a CPU write will have no effect on the field.

12.5.5.2.1   SCBResetN

Table 39. SCBResetN register format

Field Name	Bit(s)	write access	Description
CTRL	0	Full	*scb_ctrl* sub-block reset. Setting this field will reset the SCB control sub-block logic, including all configuration registers. 0 = reset 1 = default state
ISI	1	Full	*scb_isi* sub-block reset. Setting this field will reset the ISI sub-block logic. 0 = reset 1 = default state
USBH	2	Full	*scb_usbh* sub-block reset. Setting this field will reset the USB host controller core and associated logic. 0 = reset 1 = default state
USBD	3	Full	*scb_usbd* sub-block reset. Setting this field will reset the USB device controller core and associated logic. 0 = reset 1 = default state

12.5.5.2.2   SCBGo

Table 40. SCBGo register format

Field Name	Bit(s)	write access	Description
CTRL	0	Full	*scb_ctrl* sub-block go. 0 = halted 1 = running
ISI	1	Full	*scb_isi* sub-block go. 0 = halted 1 = running

12.5.5.2.3   SCBWakeUpEn

5   This register is used to gate the propagation of the USB and ISI reset signals to the CPR block.

Table 41. SCBWakeUpEn register format

Field Name	Bit(s)	write access	Description
USBWakeUpEn	0	Full	*usb_cpr_reset_n* propagation enable. 1 = enable 0 = disable
ISIWakeUpEn	1	Full	*isi_cpr_reset_n* propagation enable.

151

			1 = enable
			0 = disable

12.5.5.2.4   SCBISITxBufferArb

This register determines which source has priority at the ISITxBuffer interface on the ISI block. When a bit is set priority is given to the relevant source. When both bits have the same value, arbitration will be performed in a round-robin manner.

Table 42. SCBISITxBufferArb register format

Field Name	Bit(s)	write access	Description
CPUPriority	0	Full	CPU priority 1 = high priority 0 = low priority
USBPriority	1	Full	USB priority 1 = high priority 0 = low priority

12.5.5.2.5   SCBDebugSel

Contains address of the register selected for debug observation as it would appear on *cpu_adr*. The contents of the selected register are output in the *scb_cpu_data* bus while *cpu_scb_sel* is low and *scb_cpu_debug_valid* is asserted to indicate the debug data is valid. It is expected that a number of pseudo-registers will be made available for debug observation and these will be outlined with the implementation details.

Table 43. SCBDebugSel register format

Field Name	Bit(s)	write access	Description
CPUAdr	11:2	Full	*cpu_adr* register address.

12.5.5.2.6   USBEPnDest

This register description applies to *USBEP0Dest, USBEP1Dest, USBEP2Dest, USBEP3Dest, USBEP4Dest*. The SCB has two routing options for each packet received, based on the *DestISIId* associated with the packets source EP:

- To the DMA Manager
- To the ISI

The SCB map therefore does not need special fields to identify the DMAChannels on the ISIMaster SoPEC as this is taken care of by the SCB hardware. Thus the *USBEP0Dest* and *USBEP1Dest* registers should be programmed with 0x20 and 0x21 (for ISI0.0 and ISI0.1) respectively to ensure data arriving on these endpoints is moved directly to DRAM.

Table 44. USBEPnDest register format

152

Field Name	Bit(s)	Write access	Description
SequenceBit	0	Full	Sequence bit for packets going from USBEPn to DestISIId.DestISISubId. Every CPU write to this register initialises the value of the sequence bit and this is subsequently updated by the ISI after every successful long packet transmission.
DestISIId	4:1	Full	Destination ISI ID. Denotes the ISIId of the target SoPEC as per Table
DestISISubId	5	Full	Destination ISI sub ID. Indicates which DMAChannel of the target SoPEC the endpoint is mapped onto: 0 = DMAChannel0 1 = DMAChannel1
ChannelEn	6	Full	Communication channel enable bit for EPn. This enables/disables the communication channel for EPn. When disabled, the SCB will not accept USB packets addressed to EPn. 0 = Channel disabled 1 = Channel enabled

If the local SoPEC is connected to an external USB host, it is recommended that the EP0 communication channel should always remain enabled and mapped to DMAChannel0 on the local SoPEC, as this is intended as the primary control communication channel between the external USB host and the local SoPEC.

A SoPEC ISIMaster should map as many USB endpoints, under the control of the external host, as are required for the multi-SoPEC system it is part of. As already mentioned this mapping may be dynamically reconfigured.

12.5.5.2.7   DMAnBottomAdr

This register description applies to *DMA0BottomAdr* and *DMA1BottomAdr*.

Table 45. DMAnBottomAdr register format

Field Name	Bit(s)	Write	Description

153

		access	
DMAnBottomAdr	21:5	Full	The 256-bit aligned DRAM address of the bottom of the circular buffer (inclusive) serviced by DMAChanneln

12.5.5.2.8 DMAnTopAdr

This register description applies to *DMA0TopAdr* and *DMA1TopAdr*.

Table 46. DMAnTopAdr register format

Field Name	Bit(s)	Write access	Description
DMAnTopAdr	21:5	Full	The 256-bit aligned DRAM address of the top of the circular buffer (inclusive) serviced by DMAChanneln

12.5.5.2.9 DMAnCurrWPtr

This register description applies to DMA0CurrWPtr and DMA1CurrWPtr.

Table 47. DMAnCurrWptr register format

Field Name	Bit(s)	Write access	Description
DMAnCurrWPtr	21:5	Full	The 256-bit aligned DRAM address of the next location DMAChannel0 will write to. This register is set by the CPU at the start of a DMA operation and dynamically updated by the DMA manager during the operation.

12.5.5.2.10 DMAnIntAdr

This register description applies to *DMA0IntAdr* and *DMA1IntAdr*.

Table 48. DMAnIntAdr register format

	Bit(s)	Write access	Description
DMAnIntAdr	21:5	Full	The 256-bit aligned DRAM address of the location that will trigger an interrupt when reached by DMAChanneln buffer.

12.5.5.2.11 DMAnMaxAdr

This register description applies to *DMA0MaxAdr* and *DMA1MaxAdr*.

Table 49. DMAnMaxAdr register format

Field Name	Bit(s)	Write access	Description
DMAnMaxAdr	21:5	Full	The 256-bit aligned DRAM address of the last free location that in the DMAChanneln circular buffer. DMAChannel0 transfers will stop when it reaches this address.

12.5.5.2.12   DMAAccessEn

This register enables DMA access for the various requestors, on a per channel basis.

Table 50. DMAAccessEn register format

Field Name	Bit(s)	Write access	Description
DMAChannel0En	0	Full	DMA Channel #0 access enable. This uni-directional write channel is used by the USBD and the ISI. 1 = enable 0 = disable
DMAChannel1En	1	Full	As per USBDISI0En.
DMAChannel2En	2	Full	DMA Channel #2 access enable. This bi-directional read/write channel is used by the USBH. 1 = enable 0 = disable

5   12.5.5.2.13   DMAStatus

The status bits are not sticky bits i.e. they reflect the 'live' status of the channel.

DMAChannelNIntAdrHit and DMAChannelNMaxAdrHit status bits may only be cleared by writing to the relevant DMAnIntAdr or DMAnMaxAdr register.

Table 51. DMAStatus register format

10

Field Name	Bit(s)	Write access	Description
DMAChannel0IntAdrHit	0	None	DMA channel #0 interrupt address hit. 1= DMAChannel0 has reached the address contained in the DMA0IntAdr register. 0 = default state
DMAChannel0MaxAdrHit	1	None	DMA channel #0 max address hit. 1 = DMAChannel0 has reached the address contained in the DMA0MaxAdr

155

			register. 0 = default state
DMAChannel1IntAdrHit	3	None	As per *DMAChannel0IntAdrHit*.
DMAChannel1MaxAdrHit	4	None	As per *DMAChannel0MaxAdrHit*.

12.5.5.2.14   DMAMask register

All bits of the *DMAMask* are both readable and writable by the CPU. The DMA manager cannot alter the value of this register. All interrupts are generated in an edge sensitive manner i.e. the DMA manager will generate a *dma_icu_irq* pulse each time a status bit goes high and its corresponding mask bit is enabled.

Table 52. DMAMask register format

Field Name	Bit(s)	Write access	Description
DMAChannel0IntAdrHitIntEn	0	Full	*DMAChannel0IntAdrHit* status interrupt enable. 1 = enable 0 = disable
DMAChannel0MaxAdrHitIntEn	1	Full	*DMAChannel0MaxAdrHit* status interrupt enable. 1 = enable 0 = disable
DMAChannel1IntAdrHitIntEn	2	Full	As per *DMAChannel0IntAdrHitIntEn*
DMAChannel1MaxAdrHitIntEn	3	Full	As per *DMAChannel0MaxAdrHitIntEn*

12.5.5.2.15   CPUISITxBuffCtrl register

Table 53. CPUISITxBuffCtrl register format

Field Name	Bit(s)	Write access	Description
PktValid	0	full	This field should be set by the CPU to indicate the validity of the *CPUISITxBuff* contents. This field will be cleared by the SCB once the contents of the *CPUISITxBuff* has been copied to the *ISITxBuff*. NOTE: The CPU should not clear this field under normal operation. If the CPU clears this field during a packet transfer to the *ISITxBuff*, the transfer will be aborted - this is not

			recommended.
			1 = valid packet.
			0 = default state.
PktDesc	3:1	full	*PktDesc* field, as per Table , of the packet contained in the *CPUISITxBuff*. The CPU is responsible for maintaining the correct sequence bit value for each *ISIId.ISISubId* channel it communicates with. Only valid when *CPU-ISITxBuffCtrl.PktValid* = 1.
DestISIId	7:4	full	Denotes the *ISIId* of the target SoPEC as per Table .
DestISISubId	8	full	Indicates which DMAChannel of the target SoPEC the packet in the *CPUISITxBuff is destined for.* 1 = DMAChannel1 0 = DMAChannel0

12.5.5.2.16 USBDIntStatus

The *USBDIntStatus* register contains status bits that are related to conditions that can cause an interrupt to the CPU, if the corresponding interrupt enable bits are set in the *USBDMask* register. The field name extension *Sticky* implies that the status condition will remain registered until cleared by a CPU write of 1 to each bit of the field.

NOTE: There is no *Ep0IrregPktSticky* field because the default control EP will frequently receive packets that are not multiples of 32 bytes during normal operation.

Table 54. USBDIntStatus register format

Field Name	Bit(s)	Write access	Description
CoreSuspendSticky	0	Clear	Device core USB suspend flag. Sticky. 1 = USB suspend state. Set when device core *udcvci_suspend* signal transitions from 1 -> 0. 0 = default value.
CoreUSBResetSticky	1	Clear	Device core USB reset flag. Sticky. 1 = USB reset. Set when device core *udcvci_reset* signal transitions from 1 -> 0. 0 = default value.
CoreUSBSOFSticky	2	Clear	Device core USB Start Of Frame (SOF) flag. Sticky. 1 = USB SOF. Set when device core

			udcvci_sof signal transitions from 1 -> 0 0 = default value.
CPUISITxBuffEmptySticky	3	Clear	CPU ISI transmit buffer empty flag. Sticky. 1 = empty. 0 = default value.
CPUEp0InBuffEmptySticky	4	Clear	CPU EP0 IN buffer empty flag. Sticky. 1 = empty. 0 = default value.
CPUEp5InBuffEmptySticky	5	Clear	CPU EP5 IN buffer empty flag. Sticky. 1 = empty. 0 = default value.
Ep0InNAKSticky	6	clear	EP0-IN NAK flag. Sticky This flag is set if the USB device core issues a read request for EP0-IN and there is not a valid packet present in the EP0-IN buffer. The core will therefore send a NAK response to the IN token that was received from external USB host. This is an indicator of any back-pressure on the USB caused by EP0-IN. 1 = NAK sent. 0 = default value
Ep5InNAKSticky	7	Clear	As per *Ep0InNAK*.
Ep0OutNAKSticky	8	Clear	EP0-OUT NAK flag. Sticky This flag is set if the USB device core issues a write request for EP0-OUT and there is no space in the OUT EP buffer for a the packet. The core will therefore send a NAK response to the OUT token that was received from external USB host. This is an indicator of any back-pressure on the USB caused by EP0-OUT. 1 = NAK sent. 0 = default value
Ep1OutNAKSticky	9	Clear	As per *Ep0OutNAK*.
Ep2OutNAKSticky	10	Clear	As per *Ep0OutNAK*.
Ep3OutNAKSticky	11	Clear	As per *Ep0OutNAK*.
Ep4OutNAKSticky	12	Clear	As per *Ep0OutNAK*.
Ep1IrregPktSticky	13	Clear	EP1-OUT irregular sized packet flag. Sticky. Indicates a packet that is not a multiple of 32

			bytes in size was received by EP1-OUT. 1 = irregular sized packet received. 0 = default value.
Ep2IrregPktSticky	14	Clear	As per *Ep1IrregPktSticky.*
Ep3IrregPktSticky	15	Clear	As per *Ep1IrregPktSticky.*
Ep4IrregPktSticky	16	Clear	As per *Ep1IrregPktSticky.*
OutBuffOverFlowSticky	17	Clear	OUT EP buffer overflow flag. Sticky. This flag is set if the USB device core attempted to write a packet of more than 64 bytes to the OUT EP buffer. This is a fatal error, suggesting a problem in the USB device IP core. The SCB will take no further action. 1 = overflow condition detected. 0 = default value.
InBuffUnderRunSticky	18	clear	IN EP buffer underrun flag. Sticky. This flag is set if the USB device core attempted to read more data than was present from the IN EP buffer. This is a fatal error, suggesting a problem in the USB device IP core. The SCB will take no further action. 1 = underrun condition detected. 0 = default value.

12.5.5.2.17 USBDISIFIFOStatus

This register contains the status of the ISI mapped OUT EP packet FIFO. This is a secondary status register and will not cause any interrupts to the CPU.

Table 55. USBDISIFIFOStatus register format

5

Field Name	Bit(s)	Write access	Description
Entry0Valid	0	none	FIFO entry #0 valid field. This flag will be set by the USBD when the USB device core indicates the validity of packet entry #0 in the FIFO. 1 = valid USB packet in ISI OUT EP buffer 0. 0 = default value.
Entry0Source	3:1	none	FIFO entry #0 source field. Contains the EP associated with packet entry #0 in the FIFO. Binary Coded Decimal. Only valid when *ISIBuff0PktValid* = 1.
Entry1Valid	4	none	As per *Entry0Valid.*

Entry1Source	7:5	none	As per *Entry0Source*.
Entry2Valid	8	none	As per *Entry0Valid*.
Entry2Source	11:9	none	As per *Entry0Source*.
Entry3Valid	12	none	As per *Entry0Valid*.
Entry3Source	15:13	none	As per *Entry0Source*.

12.5.5.2.18   USBDDMA0FIFOStatus

This register description applies to *USBDDMA0FIFOStatus* and *USBDDMA1FIFOStatus*.

This register contains the status of the DMAChannelN mapped OUT EP packet FIFO. This is a secondary status register and will not cause any interrupts to the CPU.

5

Table 56. USBDDMANFIFOStatus register format

Field Name	Bit(s)	Write access	Description
Entry0Valid	0	none	FIFO entry #0 valid field. This flag will be set by the USBD when the USB device core indicates the validity of packet entry #0 in the FIFO. 1 = valid USB packet in ISI OUT EP buffer 0. 0 = default value.
Entry0Source	3:1	none	FIFO entry #0 source field. Contains the EP associated with packet entry #0 in the FIFO. Binary Coded Decimal. Only valid when *Entry0Valid* = 1.
Entry1Valid	4	none	As per *Entry0Valid*.
Entry1Source	7:5	none	As per *Entry0Source*.

12.5.5.2.19   USBDResume

This register causes the USB device core to initiate *resume* signalling to the external USB host.

Only applicable when the device core is in the *suspend* state.

10   Table 57. USBDResume register format

Field Name	Bit(s)	Write access	Description
USBDResume	0	full	USBD core resume register. The USBD will clear this register upon resume notification from the device core. 1 = generate resume signalling. 0 = default value.

12.5.5.2.20   USBDSetup

This register controls the general setup/configuration of the USBD.

Table 58. USBDSetup register format

15

Field Name	Bit(s)	write access	Description
Ep1IrregPktCntrl	0	full	EP 1 OUT irregular sized packet control. An irregular sized packet is defined as a packet that is not a multiple of 32 bytes. 1 = discard irregular sized packets. 0 = read 32 bytes from buffer, regardless of packet size.
Ep2IrregPktCntrl	1	full	As per Ep1IrregPktDiscard
Ep3IrregPktCntrl	2	full	As per Ep1IrregPktDiscard
Ep4IrregPktCntrl	3	full	As per Ep1IrregPktDiscard

12.5.5.2.21   USBDEpNInBuffCtrl register

This register description applies to *USBDEp0InBuffCtrl* and *USBDEp5InBuffCtrl*.

Table 59.   USBDEpNInBuffCtrl register format

Field Name	Bit(s)	Write access	Description
PktValid	0	full	Setting this register validates the contents of *USBDEpNInBuff*. This field will be cleared by the SCB once the packet has been successfully transmitted to the external USB host. NOTE: The CPU should not clear this field under normal operation. If the CPU clears this field during a packet transfer to the *USB*, the transfer will be aborted - this is not recommended. 1 = valid packet. 0 = default state.

5   12.5.5.2.22   USBDMask

This register serves as an interrupt mask for all USBD status conditions that can cause a CPU interrupt. Setting a field enables interrupt generation for the associated status event. Clearing a field disables interrupt generation for the associated status event. All interrupts will be generated in an edge sensitive manner, i.e. when the associated status register transitions from 0 -> 1.

10   Table 60. USBDMask register format

Field Name	Bit(s)	Write access	Description
CoreSuspendStickyEn	0	full	*CoreSuspendSticky* status interrupt enable.
CoreUSBResetStickyEn	1	full	*CoreUSBResetSticky* status interrupt enable.
CoreUSBSOFStickyEn	2	full	*CoreUSBSOFSticky* status interrupt enable.
CPUISITxBuffEmptyStickyEn	3	full	*CPUISITxBuffEmptySticky* status interrupt enable.
CPUEp0InBuffEmptyStickyEn	4	full	*CPUEp0InBuffEmptySticky* status interrupt enable.
CPUEp5InBuffEmptyStickyEn	5	full	*CPUEp5InBuffEmptySticky* status interrupt enable.

Ep0InNAKStickyEn	6	full	*Ep0InNAKSticky* status interrupt enable.
Ep5InNAKStickyEn	7	full	*Ep5InNAKSticky* status interrupt enable.
Ep0OutNAKStickyEn	8	full	*Ep0OutNAKSticky* status interrupt enable.
Ep1OutNAKStickyEn	9	full	*Ep1OutNAKSticky* status interrupt enable.
Ep2OutNAKStickyEn	10	full	*Ep2OutNAKSticky* status interrupt enable.
Ep3OutNAKStickyEn	11	full	*Ep3OutNAKSticky* status interrupt enable.
Ep4OutNAKStickyEn	12	full	*Ep4OutNAKSticky* status interrupt enable.
Ep1IrregPktStickyEn	13	full	*Ep1IrregPktSticky* status interrupt enable.
Ep2IrregPktStickyEn	14	full	*Ep2IrregPktSticky* status interrupt enable.
Ep3IrregPktStickyEn	15	full	*Ep3IrregPktSticky* status interrupt enable.
Ep4IrregPktStickyEn	16	full	*Ep4IrregPktSticky* status interrupt enable.
OutBuffOverFlowStickyEn	17	full	*OutBuffOverFlowSticky* status interrupt enable.
InBuffUnderRunStickyEn	18	full	*InBuffUnderRunSticky* status interrupt enable.

12.5.5.2.23   USBDDebug

This register is intended for debug purposes only. Contains non-sticky versions of all interrupt
capable status bits, which are referred to as *dynamic* in the table.

Table 61. USBDDebug register format

Field Name	Bit(s)	write access	Description
CoreTimeStamp	10:0	none	USB device core frame number.
CoreSuspend	11	none	Dynamic version of *CoreSuspendSticky*.
CoreUSBReset	12	none	Dynamic version of *CoreUSBResetSticky*.
CoreUSBSOF	13	none	Dynamic version of *CoreUSBSOFSticky*.
CPUISITxBuffEmpty	14	none	Dynamic version of *CPUISITxBuffEmptySticky*.
CPUEp0InBuffEmpty	15	none	Dynamic version of *CPUEp0InBuffEmptySticky*.
CPUEp5InBuffEmpty	16	none	Dynamic version of *CPUEp5InBuffEmptySticky*.
Ep0InNAK	17	none	Dynamic version of *Ep0InNAKSticky*.
Ep5InNAK	18	none	Dynamic version of *Ep5InNAKSticky*.
Ep0OutNAK	19	none	Dynamic version of *Ep0OutNAKSticky*.
Ep1OutNAK	20	none	Dynamic version of *Ep1OutNAKSticky*.
Ep2OutNAK	21	none	Dynamic version of *Ep2OutNAKSticky*.
Ep3OutNAK	22	none	Dynamic version of *Ep3OutNAKSticky*.
Ep4OutNAK	23	none	Dynamic version of *Ep4OutNAKSticky*.
Ep1IrregPkt	24	none	Dynamic version of *Ep1IrregPktSticky*.
Ep2IrregPkt	25	none	Dynamic version of *Ep2IrregPktSticky*.
Ep3IrregPkt	26	none	Dynamic version of *Ep3IrregPktSticky*.
Ep4IrregPkt	27	none	Dynamic version of *Ep4IrregPktSticky*.

OutBuffOverFlow	28	none	Dynamic version of *OutBuffOverFlowSticky*.
InBuffUnderRun	29	none	Dynamic version of *InBuffUnderRunSticky*.

12.5.5.2.24   USBHStatus

This register contains all status bits associated with the USBH. The field name extension *Sticky* implies that the status condition will remain registered until cleared by a CPU write.

Table 62. USBHStatus register format

5

Field Name	Bit(s)	Write access	Description
CoreIRQSticky	0	clear	HC core IRQ interrupt flag. Sticky Set when HC core *UHOSTC_IrqN* output signal transitions from 0 -> 1. Refer to OHCI spec for details on HC interrupt processing. 1 = IRQ interrupt from core. 0 = default value.
CoreSMISticky	1	clear	HC core SMI interrupt flag. Sticky Set when HC core *UHOSTC_SmiN* output signal transitions from 0 -> 1. Refer to OHCI spec for details on HC interrupt processing. 1 = SMI interrupt from HC. 0 = default value.
CoreBuffAcc	2	none	HC core buffer access flag. HC core *UHOSTC_BufAcc* output signal. Indicates whether the HC is accessing a descriptor or a buffer in shared system memory. 1 = buffer access 0 = descriptor access.

12.5.5.2.25   USBHMask

This register serves as an interrupt mask for all USBH status conditions that can cause a CPU interrupt. All interrupts will be generated in an edge sensitive manner, i.e. when the associated status register transitions from 0 -> 1.

10   Table 63. USBHMask register format

Field Name	Bit(s)	Write access	Description
CoreIRQIntEn	0	full	*CoreIRQSticky* status interrupt enable. 1 = enable. 0 = disable.
CoreSMIIntEn	1	full	*CoreSMISticky* status interrupt enable. 1 = enable.

			0 = disable.

12.5.5.2.26 USBHDebug

This register is intended for debug purposes only. Contains non-sticky versions of all interrupt capable status bits, which are referred to as *dynamic* in the table.

Table 64. USBHDebug register format

Field Name	Bit(s)	write access	Description
CoreIRQ	0	none	Dynamic version of *CoreIRQSticky*.
CoreSMI	1	None	Dynamic version of *CoreSMISticky*.

12.5.5.2.27 ISICntrl

This register controls the general setup/configuration of the ISI.

Note that the reset value of this register allows the SoPEC to automatically become an ISIMaster (*AutoMasterEnable* = 1) if any USB packets are received on endpoints 2-4. On becoming an ISIMaster the *ISIMasterSel* bit is set and any USB or CPU packets destined for other ISI devices are transmitted. The CPU can override this capability at any time by clearing the *AutoMasterEnable* bit.

Table 65. ISICntrl register format

Field Name	Bit(s)	Write access	Description
TxEnable	0	Full	ISI transmit enable. Enables ISI transmission of long or ping packets. ACKs may still be transmitted when this bit is 0. This is cleared by transmit errors and needs to be restarted by the CPU. 1 = Transmission enabled 0 = Transmission disabled
RxEnable	1	Full	ISI receive enable. Enables ISI reception. This is can only be cleared by the CPU and it is only anticipated that reception will be disabled when the ISI in not in use and the ISI pins are being used by the GPIO for another purpose. 1 = Reception enabled 0 = Reception disabled
ISIMasterSel	2	Full	ISI master select. Determines whether the SoPEC is an ISIMaster or not 1 = ISIMaster 0 = ISISlave
AutoMasterEnable	3	Full	ISI auto master enable.

			Enables the device to automatically become the ISIMaster if activity is detected on USB endpoints2-4. 1 = auto-master operation enabled 0 = auto-master operation disabled

12.5.5.2.28   ISIId

Table 66. ISIId register format

Field Name	Bit(s)	Write access	Description
ISIId	3:0	Full	ISIId for this SoPEC. SoPEC resets to being an ISISlave with ISIId0. 0xF (the broadcast ISIId) is an illegal value and should not be written to this register.

12.5.5.2.29   ISINumRetries

Table 67. ISINumRetries register format

Field Name	Bit(s)	Write access	Description
ISINumRetries	3:0	Full	Number of ISI retransmissions to attempt in response to an inferred NAK before aborting a long packet transmission

12.5.5.2.30   ISIPingScheduleN

This register description applies to *ISIPingSchedule0*, *ISIPingSchedule1* and *ISIPingSchedule2*.

Table 68. ISIPingScheduleN register format

Field Name	Bit(s)	Write access	Description
ISIPingSchedule	14:0	Full	Denotes which ISIIds will be receive ping packets. Note that bit0 refers to ISIId0, bit1 to ISIId1...bit14 to ISIId14.

12.5.5.2.31   ISITotalPeriod

Table 69. ISITotalPeriod register format

Field Name	Bit(s)	Write access	Description
ISITotalPeriod	3:0	Full	Reload value of the *ISITotalPeriod* counter

12.5.5.2.32   ISILocalPeriod

Table 70. ISILocalPeriod register format

Field Name	Bit(s)	Write access	Description
ISILocalPeriod	3:0	Full	Reload value of the *ISILocalPeriod* counter

12.5.5.2.33   ISIIntStatus

The *ISIIntStatus* register contains status bits that are related to conditions that can cause an interrupt to the CPU, if the corresponding interrupt enable bits are set in the *ISIMask* register.

Table 71. ISIIntStatus register

5

Field Name	Bit(s)	Write access	Description
TxErrorSticky	0	None	ISI transmit error flag. Sticky. Receiving ISI device would not accept the transmitted packet. Only set after *NumRetries* unsuccessful retransmissions. (excluding ping packets). This bit is cleared by the ISI after transmission has been re-enabled by the CPU setting the *TxEnable* bit of the *ISICntrl* register. 1= transmit error. 0 = default state.
RxFrameErrorSticky	1	Clear	ISI receive framing error flag. Sticky. This bit is set by the ISI when a framing error detected in the received packet, which can be caused by an incorrect *Start* or *Stop* field or by bit stuffing errors. 1 = framing error detected. 0 = default state.
RxCRCErrorSticky	2	Clear	ISI receive CRC error flag. This bit is set by the ISI when a CRC error is detected in an incoming packet. Other than dropping the errored packet ISI reception is unaffected by a CRC Error. 1 = CRC error 0 = default state.
RxBuffOverFlowSticky	3	Clear	ISI receive buffer over flow flag. Sticky. An overflow has occurred in the ISI receive buffer and a packet had to be dropped. 1 = over flow condition detected. 0 = default state.

12.5.5.2.34   ISITxBuffStatus

The *ISITxBuffStatus* register contains status bits that are related to the ISI Tx buffer. This is a secondary status register and will not cause any interrupts to the CPU.

Table 72. ISITxBuffStatus register format

Field Name	Bit(s)	Write access	Description
Entry0PktValid	0	None	ISI Tx buffer entry #0 packet valid flag. This flag will be set by the ISI when a valid ISI packet is written to entry #0 in the *ISITxBuff* for transmission over the ISI bus. A Tx packet is considered valid when it is 32 bytes in size and the ISI has written the packet header information to *Entry0PktDesc*, *Entry0DestISIId* and *Entry0DestISISubId*. 1 = packet valid. 0 = default value.
Entry0PktDesc	3:1	None	ISI Tx buffer entry #0 packet descriptor. PktDesc field as per Table for the packet entry #0 in the *ISITxBuff*. Only valid when *Entry0PktValid* = 1.
Entry0DestISIId	7:4	None	ISI Tx buffer entry #0 destination ISI ID. Denotes the ISIId of the target SoPEC as per Table . Only valid when *Entry0PktValid* = 1.
Entry0DestISISubId	8	None	ISI Tx buffer entry #0 destination ISI sub ID. Indicates which DMAChannel on the target SoPEC that packet entry #0 in the *ISITxBuff* is destined for. Only valid when *Entry0PktValid* = 1. 1 = DMAChannel1 0 = DMAChannel0
Entry1PktValid	9	None	As per *Entry0PktValid*.
Entry1PktDesc	12:10	None	As per *Entry0PktDesc*.
Entry1DestISIId	16:13	None	As per *Entry0DestISIId*.
Entry1DestISISubId	17	None	As per *Entry0DestISISubId*.

12.5.5.2.35   ISIRxBuffStatus

The *ISIRxBuffStatus* register contains status bits that are related to the ISI Rx buffer. This is a secondary status register and will not cause any interrupts to the CPU.

5   Table 73. ISIRxBuffStatus register format

Field Name	Bit(s)	Write access	Description
Entry0PktValid	0	None	ISI Rx buffer entry #0 packet valid flag. This flag will be set by the ISI when a valid ISI packet is received and written to entry #0 of the *ISIRxBuff*. A Rx packet is considered valid when it is 32 bytes in size and no framing or CRC errors were detected.

167

			1 = valid packet
			0 = default value
Entry0PktDesc	3:1	None	ISI Rx buffer entry #0 packet descriptor.
			PktDesc field as per Table for packet entry #0 of the
			ISIRxBuff. Only valid when *Entry0PktValid* = 1.
Entry0DestISIId	7:4	None	ISI Rx buffer 0 destination ISI ID.
			Denotes the ISIId of the target SoPEC as per Table . This
			should always correspond to the local SoPEC ISIId. Only
			valid when *Entry0PktValid* = 1.
Entry0DestISISubId	8	None	ISI Rx buffer 0 destination ISI sub ID.
			Indicates which DMAChannel on the target SoPEC that
			entry #0 of the *ISIRxBuff* is destined for. Only valid when
			Entry0PktValid = 1.
			1 = DMAChannel1
			0 = DMAChannel0
Entry1PktValid	9	None	As per *Entry0PktValid*.
Entry1PktDesc	12:10	None	As per *Entry0PktDesc*.
Entry1DestISIId	16:13	None	As per *Entry0DestISIId*.
Entry1DestISISubId	17	None	As per *Entry0DestISISubId*.

12.5.5.2.36   ISIMask register

An interrupt will be generated in an edge sensitive manner i.e. the ISI will generate an *isi_icu_irq*
pulse each time a status bit goes high and the corresponding bit of the *ISIMask* register is enabled.

Table 74. ISIMask register

5

Field Name	Bit(s)	Write access	Description
TxErrorIntEn	0	Full	*TxErrorSticky* status interrupt enable.
			1 = enable.
			0 = disable.
RxFrameErrorIntEn	1	Full	*RxFrameErrorSticky* status interrupt enable.
			1 = enable.
			0 = disable.
RxCRCErrorIntEn	2	Full	*RxCRCErrorSticky* status interrupt enable.
			1 = enable.
			0 = disable.
RxBuffOverFlowIntEn	3	Full	*RxBuffOverFlowSticky* status interrupt enable.
			1 = enable.
			0 = disable.

12.5.5.2.37 ISISubIdNSeq

This register description applies to *ISISubId0Seq* and *ISISubId0Seq*.

Table 75. ISISubIdNSeq register format

Field Name	Bit(s)	Write access	Description
ISISubIdNSeq	0	Full	ISI sub ID channel N sequence bit. This bit may be initialised by the CPU but is updated by the ISI each time an error-free long packet is received.

12.5.5.2.38 ISISubIdSeqMask

Table 76. ISISubIdSeqMask register format

Field Name	Bit(s)	Write access	Description
ISISubIdSeq0Mask	0	Full	ISI sub ID channel 0 sequence bit mask. Setting this bit ensures that the sequence bit will be ignored for incoming packets for the ISISubId. 1 = ignore sequence bit. 0 = default state.
ISISubIdSeq1Mask	1	Full	As per *ISISubIdSeq0Mask*.

12.5.5.2.39 ISINumPins

Table 77. ISINumPins register format

Field Name	Bit(s)	Write access	Description
ISINumPins	0	Full	Select number of active ISI pins. 1 = 4 pins 0 = 2 pins

12.5.5.2.40 ISITurnAround

The ISI bus turnaround time will reset to its maximum value of 0xF to provide a safer starting mode for the ISI bus. This value should be set to a value that is suitable for the physical implementation of the ISI bus, i.e. the lowest turn around time that the physical implementation will allow without significant degradation of signal integrity.

Table 78. ISITurnAround register format

Field Name	Bit(s)	Write access	Description
ISITurnAround	3:0	Full	ISI bus turn around time in ISI clock cycles (32MHz).

12.5.5.2.41 ISIShortReplyWin

The ISI short packet reply window time will reset to its maximum value of 0x1F to provide a safer starting mode for the ISI bus. This value should be set to a value that will allow for expected frequency of bit stuffing and receiver response timing.

Table 79. ISIShortReplyWin register format

Field Name	Bit(s)	Write access	Description
ISIShortReplyWin	4:0	Full	ISI long packet reply window in ISI clock cycles (32MHz).

12.5.5.2.42   ISILongReplyWin

The ISI long packet reply window time will reset to its maximum value of 0x1FF to provide a safer starting mode for the ISI bus. This value should be set to a value that will allow for expected frequency of bit stuffing and receiver response timing.

Table 80. ISILongReplyWin register format

Field Name	Bit(s)	Write access	Description
ISILongReplyWin	8:0	Full	ISI long packet reply window in ISI clock cycles (32MHz).

12.5.5.2.43   ISIDebug

This register is intended for debug purposes only. Contains non-sticky versions of all interrupt capable status bits, which are referred to as *dynamic* in the table.

Table 81. ISIDebug register format

Field Name	Bit(s)	Write access	Description
TxError	0	None	Dynamic version of *TxErrorSticky*.
RxFrameError	1	None	Dynamic version of *RxFrameErrorSticky*.
RxCRCError	2	None	Dynamic version of *RxCRCErrorSticky*.
RxBuffOverFlow	3	None	Dynamic version of *RxBuffOverFlowSticky*.

12.5.5.3 CPU Bus Interface

12.5.5.4 Control Core Logic

12.5.5.5 DIU Bus Interface

12.6   DMA REGS

All of the circular buffer registers are 256-bit word aligned as required by the DIU. The *DMAnBottomAdr* and *DMAnTopAdr* registers are inclusive i.e. the addresses contained in those registers form part of the circular buffer. The *DMAnCurrWPtr* always points to the next location the DMA manager will write to so interrupts are generated whenever the DMA manager reaches the address in either the *DMAnIntAdr* or *DMAnMaxAdr* registers rather than when it actually writes to these locations. It therefore can not write to the location in the *DMAnMaxAdr* register.

SCB Map regs

The SCB map is configured by mapping a USB endpoint on to a data sink. This is performed on a endpoint basis i.e. each endpoint has a configuration register to allow its data sink be selected. Mapping an endpoint on to a data sink does not initiate any data flow - each endpoint/data sink needs to be enabled by writing to the appropriate configuration registers for the USBD, ISI and DMA manager.

13. General Purpose IO (GPIO)

13.1   OVERVIEW

The General Purpose IO block (GPIO) is responsible for control and interfacing of GPIO pins to the rest of the SoPEC system. It provides easily programmable control logic to simplify control of GPIO functions. In all there are 32 GPIO pins of which any pin can assume any output or input function. Possible output functions are

- 4 Stepper Motor control Outputs
- 12 Brushless DC Motor Control Output (total of 2 different controllers each with 6 outputs)
- 4 General purpose high drive pulsed outputs capable of driving LEDs.
- 4 Open drain IOs used for LSS interfaces
- 4 Normal drive low impedance IOs used for the ISI interface in Multi-SoPEC mode

Each of the pins can be configured in either input or output mode, each pin is independently controlled. A programmable de-glitching circuit exists for a fixed number of input pins. Each input is a schmidt trigger to increase noise immunity should the input be used without the de-glitch circuit.

The mapping of the above functions and their alternate use in a slave SoPEC to GPIO pins is shown in Table 82 below.

Table 82. GPIO pin type

GPIO pin(s)	Pin IO Type	Default Function
gpio[3:0]	Normal drive, low impedance IO (35 Ohm), Integrated pull-up resistor	Pins 1 and 0 in ISI Mode, pins 2 and 3 in input mode
gpio[7:4]	High drive, normal impedance IO (65 Ohm), intended for LED drivers	Input Mode
gpio[31:8]	Normal drive, normal impedance IO (65 Ohm), no pull-up	Input Mode

13.2   Stepper Motor control

The motor control pins can be directly controlled by the CPU or the motor control logic can be used to generate the phase pulses for the stepper motors. The controller consists of two central counters from which the control pins are derived. The central counters have several registers (see Table) used to configure the cycle period, the phase, the duty cycle, and counter granularity.

There are two motor master counters (0 and 1) with identical features. The period of the master counters are defined by the *MotorMasterClkPeriod[1:0]* and *MotorMasterClkSrc* registers i.e. both master counters are derived from the same *MotorMasterClkSrc*. The *MotorMasterClkSrc* defines

the timing pulses used by the master counters to determine the timing period. The *MotorMasterClkSrc* can select clock sources of 1μs,100μs,10ms and *pclk* timing pulses.

The *MotorMasterClkPeriod[1:0]* registers are set to the number of timing pulses required before the timing period re-starts. Each master counter is set to the relevant *MotorMasterClkPeriod* value and

5 counts down a unit each time a timing pulse is received.

The master counters reset to *MotorMasterClkPeriod* value and count down. Once the value hits zero a new value is reloaded from the *MotorMasterClkPeriod[1:0]* registers. This ensures that no master clock glitch is generated when changing the clock period.

Each of the IO pins for the motor controller are derived from the master counters. Each pin has

10 independent configuration registers. The *MotorMasterClkSelect[3:0]* registers define which of the two master counters to use as the source for each motor control pin. The master counter value is compared with the configured *MotorCtrlLow* and *MotorCtrlHigh* registers (bit fields of the *MotorCtrlConfig* register). If the count is equal to *MotorCtrlHigh* value the motor control is set to 1, if the count is equal to *MotorCtrlLow* value the motor control pin is set to 0.

15 This allows the phase and duty cycle of the motor control pins to be varied at *pclk* granularity.

The motor control generators keep a working copy of the *MotorCtrlLow, MotorCtrlHigh* values and update the configured value to the working copy when it is safe to do so. This allows the phase or duty cycle of a motor control pin to be safely adjusted by the CPU without causing a glitch on the output pin.

20 Note that when reprogramming the *MotorCtrlLow, MotorCtrlHigh* registers to reorder the sequence of the transition points (e.g changing from low point less than high point to low point greater than high point and vice versa) care must still taken to avoid introducing glitching on the output pin.

13.3   LED CONTROL

LED lifetime and brightness can be improved and power consumption reduced by driving the LEDs

25 with a pulsed rather than a DC signal. The source clock for each of the LED pins is a 7.8kHz (128μs period) clock generated from the 1μs clock pulse from the Timers block. The *LEDDutySelect* registers are used to create a signal with the desired waveform. Unpulsed operation of the LED pins can be achieved by using CPU IO direct control, or setting *LEDDutySelect* to 0. By default the LED pins are controlled by the LED control logic.

30 13.4   LSS INTERFACE VIA GPIO

In some SoPEC system configurations one or more of the LSS interfaces may not be used. Unused LSS interface pins can be reused as general IO pins by configuring the *IOModeSelect* registers. When a mode select register for a particular GPIO pin is set to 23,22,21,20 the GPIO pin is connected to LSS control IOs 3 to 0 respectively.

35 13.5   ISI INTERFACE VIA GPIO

In Multi-SoPEC mode the SCB block (in particular the ISI sub-block) requires direct access to and from the GPIO pins. Control of the ISI interface pins is determined by the *IOModeSelect* registers. When a mode select register for a particular GPIO pin is set to 27,26,25,24 the GPIO pin connected to the ISI control bits 3 to 0 respectively. By default the GPIO pins 1 to 0 are directly controlled by

40 the ISI block.

172

In single SoPEC systems the pins can be re-used by the GPIO.

13.6 CPU GPIO CONTROL

The CPU can assume direct control of any (or all) of the IO pins individually. On a per pin basis the CPU can turn on direct access to the pin by configuring the *IOModeSelect* register to CPU direct mode. Once set the IO pin assumes the direction specified by the *CpuIODirection* register. When in output mode the value in register *CpuIOOut* will be directly reflected to the output driver. When in input mode the status of the input pin can be read by reading *CpuIOIn* register. When writing to the *CpuIOOut* register the value being written is XORed with the current value in *CpuIOOut*. The CPU can also read the status of the 10 selected de-glitched inputs by reading the *CpuIOInDeGlitch* register.

13.7 PROGRAMMABLE DE-GLITCHING LOGIC

Each IO pin can be filtered through a de-glitching logic circuit, the pin that the de-glitching logic is connected to is configured by the *InputPinSelect* registers. There are 10 de-glitching circuits, so a maximum of 10 input pin can be de-glitched at any time.

The de-glitch circuit can be configured to sample the IO pin for a predetermined time before concluding that a pin is in a particular state. The exact sampling length is configurable, but each de-glitch circuit must use one of two possible configured values (selected by *DeGlitchSelect*). The sampling length is the same for both high and low states. The *DeGlitchCount* is programmed to the number of system time units that a state must be valid for before the state is passed on. The time units are selected by *DeGlitchClkSel* and can be one of 1µs,100µs,10ms and *pclk* pulses.

For example if *DeGlitchCount* is set to 10 and *DeGlitchClkSel* set to 3, then the selected input pin must consistently retain its value for 10 system clock cycles (*pclk*) before the input state will be propagated from *CpuIOIn* to *CpuIOInDeglitch*.

13.8 INTERRUPT GENERATION

Any of the selected input pins (selected by *InputPinSelect*) can generate an interrupt from the raw or deglitched version of the input pin. There are 10 possible interrupt sources from the GPIO to the interrupt controller, one interrupt per input pin. The *InterruptSrcSelect* register determines whether the raw input or the deglitched version is used as the interrupt source.

The interrupt type, masking and priority can be programmed in the interrupt controller.

13.9 FREQUENCY ANALYSER

The frequency analyser measures the duration between successive positive edges on a selected input pin (selected by *InputPinSelect*) and reports the last period measured (*FreqAnaLastPeriod*) and a running average period (*FreqAnaAverage*).

The running average is updated each time a new positive edge is detected and is calculated by

FreqAnaAverage = (FreqAnaAverage / 8) * 7 + FreqAnaLastPeriod / 8.

The analyser can be used with any selected input pin (or its deglitched form), but only one input at a time can be selected. The input is selected by the *FreqAnaPinSelect* (range of 0 to 9) and its deglitched form can be selected by *FreqAnaPinFormSelect*.

13.10 BRUSHLESS DC (BLDC) MOTOR CONTROLLERS

The GPIO contains 2 brushless DC (BLDC) motor controllers. Each controller consists of 3 hall inputs, a direction input, and six possible outputs. The outputs are derived from the input state and a pulse width modulated (PWM) input from the Stepper Motor controller, and is given by the truth table in Table 83.

5   Table 83. Truth Table for BLDC Motor Controllers

direction	hc	hb	ha	q6	q5	q4	q3	q2	q1
0	0	0	1	0	0	0	1	PWM	0
0	0	1	1	PWM	0	0	1	0	0
0	0	1	0	PWM	0	0	0	0	1
0	1	1	0	0	0	PWM	0	0	1
0	1	0	0	0	1	PWM	0	0	0
0	1	0	1	0	1	0	0	PWM	0
0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0
1	0	0	1	0	0	PWM	0	0	1
1	0	1	1	PWM	0	0	0	0	1
1	0	1	0	PWM	0	0	1	0	0
1	1	1	0	0	0	0	1	PWM	0
1	1	0	0	0	1	0	0	PWM	0
1	1	0	1	0	1	PWM	0	0	0
1	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0

All inputs to a BLDC controller must be de-glitched. Each controller has its inputs hardwired to de-glitch circuits. Controller 1 hall inputs are de-glitched by circuits 2 to 0, and its direction input is de-

10   glitched by circuit 3. Controller 2 inputs are de-glitched by circuits 6 to 4 for hall inputs and 7 for direction input.

Each controller also requires a PWM input. The stepper motor controller outputs are reused, output 0 is connected to BLDC controller 1, and output 1 to BLDC controller 2.

The controllers have two modes of operation, internal and external direction control (configured by

15   BLDCMode). If a controller is in external direction mode the direction input is taken from a de-glitched circuit, if it is in internal direction mode the direction input is configured by the BLDCDirection register.

The BLDC controller outputs are connected to the GPIO output pins by configuring the IOModeSelect register for each pin. e.g Setting the mode register to 8 will connect q1 Controller 1 to

20   drive the pin.

13.11   IMPLEMENTATION

174

13.11.1 Definitions of I/O

Table 84. I/O definition

Port name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
tim_pulse[2:0]	3	In	Timers block generated timing pulses. 0 - 1 µs pulse 1 - 100 µs pulse 2 - 10 ms pulse
CPU Interface			
cpu_adr[8:2]	8	In	CPU address bus. Only 7 bits are required to decode the address space for this block
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
gpio_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_gpio_sel	1	In	Block select from the CPU. When *cpu_gpio_sel* is high both *cpu_adr* and *cpu_dataout* are valid
gpio_cpu_rdy	1	Out	Ready signal to the CPU. When *gpio_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the GPIO block and for a read cycle this means the data on *gpio_cpu_data* is valid.
gpio_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
gpio_cpu_debug_valid	1	Out	Debug Data valid on *gpio_cpu_data* bus. Active high
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
IO Pins			
gpio_o[31:0]	32	Out	General purpose IO output to IO driver
gpio_i[31:0]	32	In	General purpose IO input from IO receiver
gpio_e[31:0]	32	Out	General purpose IO output control. Active high driving
GPIO to LSS			

lss_gpio_dout[1:0]	2	In	LSS bus data output
			Bit 0 - LSS bus 0
			Bit 1 - LSS bus 1
gpio_lss_din[1:0]	2	Out	LSS bus data input
			Bit 0 - LSS bus 0
			Bit 1 - LSS bus 1
lss_gpio_e[1:0]	2	In	LSS bus data output enable, active high
			Bit 0 - LSS bus 0
			Bit 1 - LSS bus 1
lss_gpio_clk[1:0]	2	In	LSS bus clock output
			Bit 0 - LSS bus 0
			Bit 1 - LSS bus 1
GPIO to ISI			
gpio_isi_din[1:0]	2	Out	Input data from IO receivers to ISI.
isi_gpio_dout[1:0]	2	In	Data output from ISI to IO drivers
isi_gpio_e[1:0]	2	In	GPIO ISI pins output enable (active high) from ISI interface
usbh_gpio_power_en	1	In	Port Power enable from the USB host core, active high
gpio_usbh_over_current	1	Out	Over current detect to the USB host core, active high
Miscellaneous			
gpio_icu_irq[9:0]	10	Out	GPIO pin interrupts
gpio_cpr_wakeup	1	Out	SoPEC wakeup to the CPR block active high.
Debug			
debug_data_out[31:0]	32	In	Output debug data to be muxed on to the GPIO pins
debug_cntrl[31:0]	32	In	Control signal for each GPIO bound debug data line indicating whether or not the debug data should be selected by the pin mux

13.11.2 Configuration registers

The configuration registers in the GPIO are programmed via the CPU interface. Refer to section 11.4.3 on page 69 for a description of the protocol and timing diagrams for reading and writing registers in the GPIO. Note that since addresses in SoPEC are byte aligned and the CPU only supports 32-bit register reads and writes, the lower 2 bits of the CPU address bus are not required to decode the address space for the GPIO. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *gpio_cpu_data*. Table 85 lists the configuration registers in the GPIO block

Table 85. GPIO Register Definition

5

10

Address GPIO_base +	Register	#bits	Reset	Description
0x000-0x07C	IOModeSelect[31:0]	32x5	See Table for default values	Specifies the mode of operation for each GPIO pin. One 5 bit bus per pin. Possible assignment values and correspond controller outputs are as follows Value - Controlled by 3 to 0 - Output, LED controller 4 to 1 7 to 4 - Output Stepper Motor control 4-1 13 to 8 - Output BLDC 1 Motor control 6-1 19 to 14 - Output BLDC 2 Motor control 6-1 23 to 20 - LSS control 4-1 27 to 24 - ISI control 4-1 28 - CPU Direct Control 29 - USB power enable output 30 - Input Mode
0x080-0xA4	InputPinSelect[9:0]	10x5	0x00	Specifies which pins should be selected as inputs. Used to select the pin source to the DeGlitch Circuits.
CPU IO Control				
0x0B0	CpuIOUserMod eMask	32	0x0000 _0000	User Mode Access Mask to CPU GPIO control register. When 1 user access is enabled. One bit per gpio pin. Enables access to *CpuIODirection*, *CpuIOOut* and *CpuIOIn* in user mode.
0x0B4	CpuIOSuperMo deMask	32	0xFFFF _FFFF	Supervisor Mode Access Mask to CPU GPIO control register. When 1 supervisor access is enabled. One bit per gpio pin. Enables access to *CpuIODirection*, *CpuIOOut* and *CpuIOIn* in supervisor mode.
0x0B8	CpuIODirection	32	0x0000 _0000	Indicates the direction of each IO pin, when controlled by the CPU 0 - Indicates Input Mode 1 - Indicates Output Mode
0x0BC	CpuIOOut	32	0x0000 _0000	Value used to drive output pin in CPU direct mode. bits31:0 - Value to drive on output GPIO pins When written to the register assumes the

				new value XORed with the current value.
0x0C0	CpuIOIn	32	External pin value	Value received on each input pin regardless of mode. Read Only register.
0x0C4	CpuDeGlitchUserModeMask	10	0x000	User Mode Access Mask to *CpuIOInDeglitch* control register. When 1 user access is enabled, otherwise bit reads as zero.
0x0C8	CpuIOInDeglitch	10	0x000	Deglitched version of selected input pins. The input pins are selected by the *InputPinSelect* register. Note that after reset this register will reflect the external pin values 256 *pclk* cycles after they have stabilized. Read Only register.
Deglitch control				
0x0D0-0x0D4	DeGlitchCount[1:0]	2x8	0xFF	Deglitch circuit sample count in *DeGlitchClkSrc* selected units.
0x0D8-0x0DC	DeGlitchClkSrc[1:0]	2x2	0x3	Specifies the unit use of the GPIO deglitch circuits: 0 - 1 µs pulse 1 - 100 µs pulse 2 - 10 ms pulse 3 - *pclk*
0x0E0	DeGlitchSelect	10	0x000	Specifies which deglitch count (*DeGlitchCount*) and unit select (*DeGlitchClkSrc*) should be used with each de-glitch circuit 0 - Specifies *DeGlitchCount[0]* and *DeGlitchClkSrc[0]* 1 - Specifies *DeGlitchCount[1]* and *DeGlitchClkSrc[1]*
Motor Control				
0x0E4	MotorCtrlUserModeEnable	1	0x0	User Mode Access enable to Motor control configuration registers. When 1 user access is enabled. Enables user access to *MotorMasterClkPeriod, MotorMasterClkSrc, MotorDutySelect, MotorPhaseSelect, MotorMasterClockEnable, Motor-MasterClkSelect, BLDCMode* and

				BLDCDirection registers
0x0E8-0x0EC	MotorMasterCl kPeriod[1:0]	2x16	0x0000	Specifies the motor controller master clock periods in MotorMasterClkSrc selected units
0x0F0	MotorMasterCl kSrc	2	0x0	Specifies the unit use by the motor controller master clock generator: 0 - 1 μs pulse 1 - 100 μs pulse 2 - 10 ms pulse 3 - pclk
0x0F4-0x100	MotorCtrlConfig [3:0]	4x32	0x0000 _0000	Specifies the transition points in the clock period for each motor control pin. One register per pin bits 15:0 - MotorCtrlLow, high to low transition point bits 31:16 - MotorCtrlHigh, low to high transition point
0x104	MotorMasterCl kSelect	4	0x0	Specifies which motor master clock should be used as a pin generator source 0 - Clock derived from MotorMasterClockPeriod[0] 1 -Clock derived from MotorMasterClockPeriod[1]
0x108	MotorMasterCl ockEnable	2	0x0	Enable the motor master clock counter. When 1 count is enabled Bit 0 - Enable motor master clock 0 Bit 1 - Enable motor master clock 1
BLDC Motor Controllers				
0x10C	BLDCMode	2	0x0	Specifies the Mode of operation of the BLDC Controller. One bit per Controller. 0- External direction control 1- Internal direction control
0x110	BLDCDirection	2	0x0	Specifies the direction input of the BLDC controller. Only used when BLDC controller is an internal direction control mode. One bit per controller.
LED control				
0x114	LEDCtrlUserMo deEnable	4	0x0	User Mode Access enable to LED control configuration registers. When 1 user access is enabled.

				One bit per *LEDDutySelect* select register.
0x118-0x124	LEDDutySelect [3:0]	4x3	0x0	Specifies the duty cycle for each LED control output. See Figure 54 for encoding details. The *LEDDutySelect[3:0]* registers determine the duty cycle of the LED controller outputs
Frequency Analyser				
0x130	FreqAnaUserModeEnable	1	0x0	User Mode Access enable to Frequency analyser configuration registers. When 1 user access is enabled. Controls access to *FreqAnaPinFormSelect*, *FreqAnaLastPeriod*, *FreqAnaAverage* and *FreqAnaCountInc*.
0x134	FreqAnaPinSelect	4	0x00	Selects which selected input should be used for the frequency analyses.
0x138	FreqAnaPinFormSelect	1	0x0	Selects if the frequency analyser should use the raw input or the deglitched form. 0 - Deglitched form of input pin 1 - Raw form of input pin
0x13C	FreqAnaLastPeriod	16	0x0000	Frequency Analyser last period of selected input pin.
0x140	FreqAnaAverage	16	0x0000	Frequency Analyser average period of selected input pin.
0x144	FreqAnaCountInc	20	0x0000 0	Frequency Analyser counter increment amount. For each clock cycle no edge is detected on the selected input pin the accumulator is incremented by this amount.
0x148	FreqAnaCount	32	0x0000 _0000	Frequency Analyser running counter (Working register)
Miscellaneous				
0x150	InterruptSrcSelect	10	0x3FF	Interrupt source select.1 bit per selected input. Determines whether the interrupt source is direct form the selected input pin or the deglitched version. Input pins are selected by the *DeGlitchPinSelect* register. 0 - Selected input direct 1 - Deglitched selected input
0x154	DebugSelect[8:2]	7	0x00	Debug address select. Indicates the address of the register to report on the

				gpio_cpu_data bus when it is not otherwise being used.
0x158-0x15C	MotorMasterCount[1:0]	2x16	0x0000	Motor master clock counter values. Bus 0 - Master clock count 0 Bus 1 - Master clock count 1 Read Only registers
0x160	WakeUpInputMask	10	0x000	Indicates which deglitched inputs should be considered to generate the CPR wakeup. Active high
0x164	WakeUpLevel	1	0	Defines the level to detect on the masked GPIO inputs to generate a wakeup to the CPR 0 - Level 0 1 - Level 1
0x168	USBOverCurrentPinSelect	4	0x00	Selects which deglitched input should be used for the USB over current detect.

13.11.2.1 Supervisor and user mode access

The configuration registers block examines the CPU access type (*cpu_acode* signal) and determines if the access is allowed to that particular register, based on configured user access registers. If an access is not allowed the GPIO will issue a bus error by asserting the *gpio_cpu_berr* signal.

All supervisor and user program mode accesses will result in a bus error.

Access to the *CpuIODirection*, *CpuIOOut* and *CpuIOIn* is filtered by the *CpuIOUserModeMask* and *CpuIOSuperModeMask* registers. Each bit masks access to the corresponding bits in the *CpuIO** registers for each mode, with *CpuIOUserModeMask* filtering user data mode access and *CpuIOSuperModeMask* filtering supervisor data mode access.

The addition of the *CpuIOSuperModeMask* register helps prevent potential conflicts between user and supervisor code read modify write operations. For example a conflict could exist if the user code is interrupted during a read modify write operation by a supervisor ISR which also modifies the *CpuIO** registers.

An attempt to write to a disabled bit in user or supervisor mode will be ignored, and an attempt to read a disabled bit returns zero. If there are no user mode enabled bits then access is not allowed in user mode and a bus error will result. Similarly for supervisor mode.

When writing to the *CpuIOOut* register, the value being written is XORed with the current value in the *CpuIOOut* register, and the result is reflected on the GPIO pins.

The pseudocode for determining access to the *CpuIOOut* register is shown below. Similar code could be shown for the *CpuIODirection* and *CpuIOIn* registers. Note that when writing to *CpuIODirection* data is deposited directly and not XORed with the existing data (as in the *CpuIOOut* case).

```
            if (cpu_acode == SUPERVISOR_DATA_MODE) then
               // supervisor mode
               if (CpuIOSuperModeMask[31:0] == 0 ) then
                  // access is denied, and bus error
                  gpio_cpu_berr = 1
               elsif (cpu_rwn == 1) then
                  // read mode (no filtering needed)
                  gpio_cpu_data[31:0] = CpuIOOut[31:0]
               else
                  // write mode,filtered by mask
                  mask[31:0]                    =      (cpu_dataout[31:0]    &
               CpuIOSuperModeMask[31:0])
                     CpuIOOut[31:0]  =  (cpu_dataout[31:0]  ^      mask[31:0]   )
               //bitwise XOR operator
            elsif (cpu_acode == USER_DATA_MODE) then
               // user datamode
               if (CpuIOUserModeMask[31:0] == 0 ) then
                  // access is denied, and bus error
                  gpio_cpu_berr = 1
               elsif (cpu_rwn == 1) then
                  // read mode, filtered by mask
                  gpio_cpu_data        =     (       CpuIOOut[31:0]        &
               CpuIOUserModeMask[31:0])
               else
                  // write mode,filtered by mask
                  mask[31:0]                    =      (cpu_dataout[31:0]    &
               CpuIOUserModeMask[31:0])
                     CpuIOOut[31:0]  =  (cpu_dataout[31:0]  ^      mask[31:0]   )
               //bitwise XOR operator
            else
               // access is denied, bus error
               gpio_cpu_berr = 1
```

Table 86 details the access modes allowed for registers in the GPIO block. In supervisor mode all registers are accessible. In user mode forbidden accesses will result in a bus error (*gpio_cpu_berr* asserted).

Table 86. GPIO supervisor and user access modes

Register Address	Registers	Access Permitted
0x000-0x07C	IOModeSelect[31:0]	Supervisor data mode only
0x080-0x94	InputPinSelect[9:0]	Supervisor data mode only
CPU IO Control		
0x0B0	CpuIOUserModeMask	Supervisor data mode only
0x0B4	CpuIOSuperModeMask	Supervisor data mode only

0x0B8	CpuIODirection	CpuIOUserModeMask and CpuIOSuperModeMask filtered
0x0BC	CpuIOOut	CpuIOUserModeMask and CpuIOSuperModeMask filtered
0x0C0	CpuIOIn	CpuIOUserModeMask and CpuIOSuperModeMask filtered
0x0C4	CpuDeGlitchUserModeMask	Supervisor data mode only
0x0C8	CpuIOInDeglitch	CpuDeGlitchUserModeMask filtered. Unrestricted Supervisor data mode access
Deglitch control		
0x0D0-0x0D4	DeGlitchCount[1:0]	Supervisor data mode only
0x0D8-0x0DC	DeGlitchClkSrc[1:0]	Supervisor data mode only
0x0E0	DeGlitchSelect	Supervisor data mode only
Motor Control		
0x0E4	MotorCtrlUserModeEnable	Supervisor data mode only
0x0E8-0x0EC	MotorMasterClkPeriod[1:0]	MotorCtrlUserModeEnable enabled.
0x0F0	MotorMasterClkSrc	MotorCtrlUserModeEnable enabled.
0x0F4-0x100	MotorCtrlConfig[3:0]	MotorCtrlUserModeEnable enabled
0x104	MotorMasterClkSelect	MotorCtrlUserModeEnable enabled
0x108	MotorMasterClockEnable	MotorCtrlUserModeEnable enabled
BLDC Motor Controllers		
0x10C	BLDCMode	MotorCtrlUserModeEnable Enabled
0x110	BLDCDirection	MotorCtrlUserModeEnable Enabled
LED control		
0x114	LEDCtrlUserModeEnable	Supervisor data mode only
0x118-0x124	LEDDutySelect[3:0]	LEDCtrlUserModeEnable[3:0] enabled
Frequency Analyser		
0x130	FreqAnaUserModeEnable	Supervisor data mode only
0x134	FreqAnaPinSelect	FreqAnaUserModeEnable enabled
0x138	FreqAnaPinFormSelect	FreqAnaUserModeEnable enabled
0x13C	FreqAnaLastPeriod	FreqAnaUserModeEnable enabled
0x140	FreqAnaAverage	FreqAnaUserModeEnable enabled
0x144	FreqAnaCountInc	FreqAnaUserModeEnable enabled
0x148	FreqAnaCount	FreqAnaUserModeEnable enabled
Miscellaneous		
0x150	InterruptSrcSelect	Supervisor data mode only

0x154	DebugSelect[8:2]	Supervisor data mode only
0x158-0x15C	MotorMasterCount[1:0]	Supervisor data mode only
0x160	WakeUpInputMask	Supervisor data mode only
0x164	WakeUpLevel	Supervisor data mode only
0x168	USBOverCurrentPinSelect	Supervisor data mode only

13.11.3 GPIO partition

13.11.4 IO control

The IO control block connects the IO pin drivers to internal signalling based on configured setup registers and debug control signals.

```
// Output Control
for (i=0; i<32 ; i++) {
if (debug_cntrl[i] == 1) then    // debug mode
  gpio_e[i] = 1;gpio_o[i] =debug_data_out[i]
else // normal mode
  case io_mode_select[i] is
    0 : gpio_e[i] =1 ;gpio_o[i] =led_ctrl[0]          // LED
output 1
    1 : gpio_e[i] =1 ;gpio_o[i] =led_ctrl[1]          // LED
output 2
    2 : gpio_e[i] =1 ;gpio_o[i] =led_ctrl[2]          // LED
output 3
    3 : gpio_e[i] =1 ;gpio_o[i] =led_ctrl[3]          // LED
output 4
    4 : gpio_e[i] =1 ;gpio_o[i] =motor_ctrl[0]     // Stepper
Motor Control 1
    5 : gpio_e[i] =1 ;gpio_o[i] =motor_ctrl[1]     // Stepper
Motor Control 2
    6 : gpio_e[i] =1 ;gpio_o[i] =motor_ctrl[2]     // Stepper
Motor Control 3
    7 : gpio_e[i] =1 ;gpio_o[i] =motor_ctrl[3]     // Stepper
Motor Control 4
    8 : gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[0][0]    // BLDC
Motor Control 1,output 1
    9 : gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[0][1]    // BLDC
Motor Control 1,output 2
    10: gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[0][2]    // BLDC
Motor Control 1,output 3
    11: gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[0][3]    // BLDC
Motor Control 1,output 4
    12: gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[0][4]    // BLDC
Motor Control 1,output 5
    13: gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[0][5]    // BLDC
Motor Control 1,output 6
```

```
        14:  gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[1][0]    // BLDC
Motor Control 2,output 1
        15:  gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[1][1]    // BLDC
Motor Control 2,output 2
        16:  gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[1][2]    // BLDC
Motor Control 2,output 3
        17:  gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[1][3]    // BLDC
Motor Control 2,output 4
        18:  gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[1][4]    // BLDC
Motor Control 2,output 5
        19:  gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[1][5]    // BLDC
Motor Control 2,output 6
        20: gpio_e[i] =1 ;gpio_o[i] =lss_gpio_clk[0]    // LSS Clk
0
        21: gpio_e[i] =1 ;gpio_o[i] =lss_gpio_clk[1]    // LSS Clk
1
        22:       gpio_e[i]       =lss_gpio_e[0]       ;gpio_o[i]
=lss_gpio_dout[0];  // LSS Data 0
        gpio_lss_din[0] = gpio_i[i]
        23:       gpio_e[i]       =lss_gpio_e[1]       ;gpio_o[i]
=lss_gpio_dout[1];  // LSS Data 1
        gpio_lss_din[1] = gpio_i[i]
        24:       gpio_e[i]       =isi_gpio_e[0]       ;gpio_o[i]
=isi_gpio_dout[0];  // ISI Control 1
        gpio_isi_din[0] = gpio_i[i]
        25:       gpio_e[i]       =isi_gpio_e[1]       ;gpio_o[i]
=isi_gpio_dout[1];  // ISI Control 2
        gpio_isi_din[1] = gpio_i[i]
        26:       gpio_e[i]       =isi_gpio_e[2]       ;gpio_o[i]
=isi_gpio_dout[2];  // ISI Control 3
        gpio_isi_din[2] = gpio_i[i]
        27:       gpio_e[i]       =isi_gpio_e[3]       ;gpio_o[i]
=isi_gpio_dout[3];  // ISI Control 4
        gpio_isi_din[3] = gpio_i[i]
        28: gpio_e[i] =cpu_io_dir[i] ;gpio_o[i] =cpu_io_out[i];
// CPU Direct
        29:    gpio e[i]    =1    ;gpio o[i]    =usbh gpio power en
// USB host power enable
        30:       gpio e[i]       =0       ;gpio o[i]       =0
// Input only mode
   end case
   // all gpio are always readable by the CPU
   cpu_io_in[i] = gpio_i[i];
   }
```

The input selection pseudocode, for determining which pin connects to which de-
glitch circuit.

```
for( i=0 ;i < 10 ; i++) {
   pin_num            = input_pin_select[i]
   deglitch_input[i] = gpio_i[pin_num]
   }
```

The *gpio_usbh_over_current* output to the USB core is driven by a selected deglitched input (configured by the *USBOverCurrentPinSelect* register).

```
index = USBOverCurrentPinSelect
gpio_usbh_over_current = cpu_io_in_deglitch[index]
```

13.11.5   Wakeup generator

The wakeup generator compares the deglitched inputs with the configured mask (WakeUpInputMask) and level (WakeUpLevel), and determines whether to generate a wakeup to the CPR block.

```
for (i =0;i<10; i++) {
   if (wakeup_level = 0) then   // level 0 active
      wakeup  =  wakeup  OR  wakeup_input_mask[i]  AND  NOT
cpu_io_in_deglitch[i]
   else                         // level 1 active
      wakeup  =  wakeup  OR  wakeup_input_mask[i]  AND
cpu_io_in_deglitch[i]
   }
// assign the output
gpio_cpr_wakeup = wakeup
```

13.11.6   LED pulse generator

The pulse generator logic consists of a 7-bit counter that is incremented on a 1µs pulse from the timers block (*tim_pulse[0]*). The LED control signal is generated by comparing the count value with the configured duty cycle for the LED (*led_duty_sel*).

The logic is given by:

```
for (i=0 i<4 ;i++) { // for each LED pin
   // period divided into 8 segments
   period_div8 = cnt[6:4];
   if (period_div8 < led_duty_sel[i]) then
      led_ctrl[i] = 1
   else
      led_ctrl[i] = 0
   }
// update the counter every 1us pulse
if (tim_pulse[0] == 1) then
      cnt ++
```

13.11.7   Stepper Motor control

The motor controller consists of 2 counters, and 4 phase generator logic blocks, one per motor control pin. The counters decrement each time a timing pulse (*cnt_en*) is received. The counters start at the configured clock period value (*motor_mas_clk_period*) and decrement to zero. If the

186

counters are enabled (via *motor_mas_clk_enable*), the counters will automatically restart at the configured clock period value, otherwise they will wait until the counters are re-enabled.

The timing pulse period is one of *pclk*, 1μs, 100μs, 1ms depending on the *motor_mas_clk_sel* signal. The counters are used to derive the phase and duty cycle of each motor control pin.

```
// decrement logic
if (cnt_en == 1) then
    if ((mas_cnt == 0) AND (motor_mas_clk_enable == 1)) then
        mas_cnt = motor_mas_clk_period[15:0]
    elsif ((mas_cnt == 0) AND (motor_mas_clk_enable == 0)) then
        mas_cnt = 0
    else
        mas_cnt --
else    // hold the value
    mas_cnt = mas_cnt
```

The phase generator block generates the motor control logic based on the selected clock generator (*motor_mas_clk_sel*) the motor control high transition point (*curr_motor_ctrl_high*) and the motor control low transition point (*curr_motor_ctrl_low*).

The phase generator maintains current copies of the *motor_ctrl_config* configuration value (*motor_ctrl_config[31:16]* becomes *curr_motor_ctrl_high* and *motor_ctrl_config[15:0]* becomes *curr_motor_ctrl_low*). It updates these values to the current register values when it is safe to do so without causing a glitch on the output motor pin.

Note that when reprogramming the *motor_ctrl_config* register to reorder the sequence of the transition points (e.g changing from low point less than high point to low point greater than high point and vice versa) care must taken to avoid introducing glitching on the output pin.

There are 4 instances one per motor control pin.

The logic is given by:

```
// select the input counter to use
if (motor_mas_clk_sel == 1) then
    count  = mas_cnt[1]
else
    count  = mas_cnt[0]
// Generate the phase and duty cycle
if (count == curr_motor_ctrl_low) then
    motor_ctrl = 0
elsif (count == curr_motor_ctrl_high) then
    motor_ctrl = 1
else
    motor_ctrl = motor_ctrl // remain the same
// update the current registers at period boundary
if (count == 0) then
```

```
                curr_motor_ctrl_high    =   motor_ctrl_config[31:16]            //
            update to new high value
                curr_motor_ctrl_low     =   motor_ctrl_config[15:0]             //
            update to new high value
```

13.11.8   Input deglitch

The input deglitch logic rejects input states of duration less than the configured number of time units (*deglitch_cnt*), input states of greater duration are reflected on the output *cpu_io_in_deglitch*. The time units used (either *pclk*, 1μs, 100μs, 1ms) by the deglitch circuit is selected by the *deglitch_clk_src* bus.

There are 2 possible sets of *deglitch_cnt* and *deglitch_clk_src* that can be used to deglitch the input pins. The values used are selected by the *deglitch_sel* signal.

There are 10 deglitch circuits in the GPIO. Any GPIO pin can be connected to a deglitch circuit. Pins are selected for deglitching by the *InputPinSelect* registers.

Each selected input can be used to generate an interrupt. The interrupt can be generated from the raw input signal (*deglitch_input*) or a deglitched version of the input (*cpu_io_in_deglitch*). The interrupt source is selected by the *interrupt_src_select* signal.

The counter logic is given by

```
if (deglitch_input != deglitch_input_delay) then
    cnt         = deglitch_cnt
    output_en = 0
elsif (cnt == 0 ) then
    cnt         = cnt
    output_en = 1
elsif (cnt_en == 1) then
    cnt --
    output_en = 0
```

13.11.9   Frequency Analyser

The frequency analyser block monitors a selected deglitched input (*cpu_io_in_deglitch*) or a direct selected input (*deglitch_input*) and detects positive edges. The selected input is configured by *FreqAnaPinSelect* and *FreqAnaPinFormSel* registers. Between successive positive edges detected on the input it increments a counter (*FreqAnaCount*) by a programmed amount (*FreqAnaCountInc*) on each clock cycle. When a positive edge is detected the *FreqAnaLastPeriod* register is updated with the top 16 bits of the counter and the counter is reset. The frequency analyser also maintains a running average of the *FreqAnaLastPeriod* register. Each time a positive edge is detected on the input the *FreqAnaAverage* register is updated with the new calculated *FreqAnaLastPeriod*. The average is calculated as 7/8 the current value plus 1/8 of the new value. The *FreqAnaLastPeriod*, *FreqAnaCount* and *FreqAnaAverage* registers can be written to by the CPU.

The pseudocode is given by

```
if ((pin == 1) AND pin_delay ==0 ))then      // positive edge
detected
```

```
freq_ana_lastperiod[15:0]  = freq_ana_count[31:16]
freq_ana_average[15:0]            =  freq_ana_average[15:0]    -
freq_ana_average[15:3]
                                                               +
freq_ana_lastperiod[15:3]
   freq_ana_count[15:0]       = 0
else
   freq_ana_count[31:0]                 =  freq_ana_count[31:0]   +
freq_ana_count_inc[19:0]
// implement the configuration register write
if (wr_last_en == 1) then
   freq_ana_lastperiod = wr_data
elsif (wr_average_en == 1 ) then
   freq_ana_average = wr_data
elsif (wr_freq_count_en == 1) then
   freq_ana_count = wr_data
```

13.11.10 BLDC Motor Controller

The BLDC controller logic is identical for both instances, only the input connections are different. The logic implements the truth table shown in Table . The six *q* outputs are combinationally based on the *direction*, *ha*, *hb*, *hc* and *pwm* inputs. The direction input has 2 possible sources selected by the mode, the pseudocode is as follows

```
// determine if in internal or external direction mode
if (mode == 1) then        // internal mode
   direction = int_direction
else                       // external mode
   direction = ext_direction
```

14 Interrupt Controller Unit (ICU)

The interrupt controller accepts up to N input interrupt sources, determines their priority, arbitrates based on the highest priority and generates an interrupt request to the CPU. The ICU complies with the interrupt acknowledge protocol of the CPU. Once the CPU accepts an interrupt (i.e. processing of its service routine begins) the interrupt controller will assert the next arbitrated interrupt if one is pending.

Each interrupt source has a fixed vector number N, and an associated configuration register, *IntReg[N]*. The format of the *IntReg[N]* register is shown in Table 87 below.

Table 87. IntReg[N] register format

Field	bit(s)	Description
Priority	3:0	Interrupt priority
Type	5:4	Determines the triggering conditions for the interrupt 00 - Positive edge 10 - Negative edge

		01 - Positive level
		11 - Negative level
Mask	6	Mask bit.
		1 - Interrupts from this source are enabled,
		0 - Interrupts from this source are disabled.
		Note that there may be additional masks in operation at the source of the interrupt.
Reserved	31:7	Reserved. Write as 0.

Once an interrupt is received the interrupt controller determines the priority and maps the programmed priority to the appropriate CPU priority levels, and then issues an interrupt to the CPU. The programmed interrupt priority maps directly to the LEON CPU interrupt levels. Level 0 is no interrupt. Level 15 is the highest interrupt level.

14.1   INTERRUPT PREEMPTION

With standard LEON pre-emption an interrupt can only be pre-empted by an interrupt with a higher priority level. If an interrupt with the same priority level (1 to 14) as the interrupt being serviced becomes pending then it is not acknowledged until the current service routine has completed.

Note that the level 15 interrupt is a special case, in that the LEON processor will continue to take level 15 interrupts (i.e re-enter the ISR) as long as level 15 is asserted on the *icu_cpu_ilevel*.

Level 0 is also a special case, in that LEON consider level 0 interrupts as no interrupt, and will not issue an acknowledge when level 0 is presented on the *icu_cpu_ilevel* bus.

Thus when pre-emption is required, interrupts should be programmed to different levels as interrupt priorities of the same level have no guaranteed servicing order. Should several interrupt sources be programmed with the same priority level, the lowest value interrupt source will be serviced first and so on in increasing order.

The interrupt is directly acknowledged by the CPU and the ICU automatically clears the pending bit of the lowest value pending interrupt source mapped to the acknowledged interrupt level.

All interrupt controller registers are only accessible in supervisor data mode. If the user code wishes to mask an interrupt it must request this from the supervisor and the supervisor software will resolve user access levels.

14.2   INTERRUPT SOURCES

The mapping of interrupt sources to interrupt vectors (and therefore *IntReg[N]* registers) is shown in Table 88 below. Please refer to the appropriate section of this specification for more details of the interrupt sources.

Table 88. Interrupt sources vector table

Vector	Source	Description
0	Timers	WatchDog Timer Update request
1	Timers	Generic Timer 1 interrupt

2	Timers	Generic Timer 2 interrupt
3	PCU	PEP Sub-system Interrupt- TE finished band
4	PCU	PEP Sub-system Interrupt- LBD finished band
5	PCU	PEP Sub-system Interrupt- CDU finished band
6	PCU	PEP Sub-system Interrupt- CDU error
7	PCU	PEP Sub-system Interrupt- PCU finished band
8	PCU	PEP Sub-system Interrupt- PCU Invalid address interrupt
9	PHI	PEP Sub-system Interrupt- PHI Line Sync Interrupt
10	PHI	PEP Sub-system Interrupt- PHI Buffer underrun
11	PHI	PEP Sub-system Interrupt- PHI Page finished
12	PHI	PEP Sub-system Interrupt- PHI Print ready
13	SCB	USB Host interrupt
14	SCB	USB Device interrupt
15	SCB	ISI interrupt
16	SCB	DMA interrupt
17	LSS	LSS interrupt, LSS interface 0 interrupt request
18	LSS	LSS interrupt, LSS interface 1 interrupt request
19-28	GPIO	GPIO general purpose interrupts
29	Timers	Generic Timer 3 interrupt

14.3   IMPLEMENTATION

14.3.1   Definitions of I/O

Table 89. Interrupt Controller Unit I/O definition

Port name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
CPU interface			
cpu_adr[7:2]	6	In	CPU address bus. Only 6 bits are required to decode the address space for the ICU block
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
icu_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_icu_sel	1	In	Block select from the CPU. When cpu_icu_sel is high both cpu_adr and cpu_dataout are valid
icu_cpu_rdy	1	Out	Ready signal to the CPU. When icu_cpu_rdy is high it indicates the last cycle of the access. For a write cycle this means cpu_dataout has been

			registered by the ICU block and for a read cycle this means the data on *icu_cpu_data* is valid.
icu_cpu_ilevel[3:0]	4	Out	Indicates the priority level of the current active interrupt.
cpu_iack	1	In	Interrupt request acknowledge from the LEON core.
cpu_icu_ilevel[3:0]	4	In	Interrupt acknowledged level from the LEON core
icu_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
icu_cpu_debug_valid	1	Out	Debug Data valid on *icu_cpu_data* bus. Active high
Interrupts			
tim_icu_wd_irq	1	In	Watchdog timer interrupt signal from the Timers block
tim_icu_irq[2:0]	3	In	Generic timer interrupt signals from the Timers block .
gpio_icu_irq[9:0]	10	In	GPIO pin interrupts
usb_icu_irq[1:0]	2	In	USB host and device interrupts from the SCB Bit 0 - USB Host interrupt Bit 1 - USB Device interrupt
isi_icu_irq	1	In	ISI interrupt from the SCB
dma_icu_irq	1	In	DMA interrupt from the SCB
lss_icu_irq[1:0]	2	In	LSS interface interrupt request
cdu_finishedband	1	In	Finished band interrupt request from the CDU
cdu_icu_jpegerror	1	In	JPEG error interrupt from the CDU
lbd_finishedband	1	In	Finished band interrupt request from the LBD
te_finishedband	1	In	Finished band interrupt request from the TE
pcu_finishedband	1	In	Finished band interrupt request from the PCU
pcu_icu_address_invalid	1	In	Invalid address interrupt request from the PCU
phi_icu_underrun	1	In	Buffer underrun interrupt request from the PHI
phi_icu_page_finish	1	In	Page finished interrupt request from the PHI
phi_icu_print_rdy	1	In	Print ready interrupt request from the PHI

phi_icu_linesync_int	1	In	Line sync interrupt request from the PHI

14.3.2   Configuration registers

The configuration registers in the ICU are programmed via the CPU interface. Refer to section 11.4 on page 69 for a description of the protocol and timing diagrams for reading and writing registers in the ICU. Note that since addresses in SoPEC are byte aligned and the CPU only supports 32-bit register reads and writes, the lower 2 bits of the CPU address bus are not required to decode the address space for the ICU. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *icu_pcu_data*. Table 90 lists the configuration registers in the ICU block.

The ICU block will only allow supervisor data mode accesses (i.e. *cpu_acode[1:0]* = *SUPERVISOR_DATA*). All other accesses will result in *icu_cpu_berr* being asserted.

Table 90. ICU Register Map

Address ICU base +	Register	#bits	Reset	Description
0x00 – 0x74	IntReg[29:0]	30x7	0x00	Interrupt vector configuration register
0x88	IntClear	30	0x0000 _0000	Interrupt pending clear register. If written with a one it clears corresponding interrupt Bits[30:0] - Interrupts sources 30 to 0 (Reads as zero)
0x90	IntPending	30	0x0000 _0000	Interrupt pending register. (Read Only) Bits[30:0]- Interrupts sources 30 to 0
0xA0	IntSource	5	0x1F	Indicates the interrupt source of the last acknowl-edged interrupt. The *NoInterrupt* value is defined as all bits set to one. (Read Only)
0xC0	DebugSelect[7:2]	6	0x00	Debug address select. Indicates the address of the register to report on the *icu_cpu_data* bus when it is not otherwise being used.

14.3.3   ICU partition

14.3.4   Interrupt detect

The ICU contains multiple instances of the interrupt detect block, one per interrupt source. The interrupt detect block examines the interrupt source signal, and determines whether it should generate request pending (*int_pend*) based on the configured interrupt type and the interrupt source conditions. If the interrupt is not masked the interrupt will be reflected to the interrupt arbiter via the *int_active* signal. Once an interrupt is pending it remains pending until the interrupt is accepted by the CPU or it is level sensitive and gets removed. Masking a pending interrupt has the effect of removing the interrupt from arbitration but the interrupt will still remain pending.

When the CPU accepts the interrupt (using the normal ISR mechanism), the interrupt controller automatically generates an interrupt clear for that interrupt source (*cpu_int_clear*). Alternatively if the interrupt is masked, the CPU can determine pending interrupts by polling the *IntPending* registers. Any active pending interrupts can be cleared by the CPU without using an ISR via the *IntClear* registers.

Should an interrupt clear signal (either from the interrupt clear unit or the CPU) and a new interrupt condition happen at the same time, the interrupt will remain pending. In the particular case of a level sensitive interrupt, if the level remains the interrupt will stay active regardless of the clear signal. The logic is shown below:

```
mask            = int_config[6]
type            = int_config[5:4]
int_pend        = last_int_pend              // the last pending
interrupt
// update the pending FF
// test for interrupt condition
if (type == NEG_LEVEL) then
   int_pend = NOT(int_src)
elsif (type == POS_LEVEL)
   int_pend = int_src
elsif ((type   ==   POS_EDGE  )  AND  (int_src   ==   1)   AND
(last_int_src == 0))
   int_pend = 1
elsif ((type   ==   NEG_EDGE  )  AND  (int_src   ==   0)   AND
(last_int_src == 1))
   int_pend = 1
elsif ((int_clear == 1 )OR (cpu_int_clear==1)) then
   int_pend = 0
else
   int_pend = last_int_pend // stay the same as before
// mask the pending bit
if (mask == 1) then
   int_active = int_pend
else
   int_active = 0
// assign the registers
last_int_src  = int_src
last_int_pend = int_pend
```

14.3.5 Interrupt arbiter

The interrupt arbiter logic arbitrates a winning interrupt request from multiple pending requests based on configured priority. It generates the interrupt to the CPU by setting *icu_cpu_ilevel* to a non-zero value. The priority of the interrupt is reflected in the value assigned to *icu_cpu_ilevel*, the higher the value the higher the priority, 15 being the highest, and 0 considered no interrupt.

```
// arbitrate with the current winner
int_ilevel      = 0
```

```
                    for (i=0;i<30;i++) {
                      if ( int_active[i] == 1) then {
                        if (int_config[i][3:0] > win_int_ilevel[3:0] ) then
                           win_int_ilevel[3:0] = int_config[i][3:0]
                          }
                       }
                     }
                    // assign the CPU interrupt level
                    int_ilevel = win_int_ilevel[3:0]
```

14.3.6   Interrupt clear unit

The interrupt clear unit is responsible for accepting an interrupt acknowledge from the CPU, determining which interrupt source generated the interrupt, clearing the pending bit for that source and updating the *IntSource* register.

When an interrupt acknowledge is received from the CPU, the interrupt clear unit searches through each interrupt source looking for interrupt sources that match the acknowledged interrupt level (*cpu_icu_ilevel*) and determines the winning interrupt (lower interrupt source numbers have higher priority). When found the interrupt source pending bit is cleared and the *IntSource* register is updated with the interrupt source number.

The LEON interrupt acknowledge mechanism automatically disables all other interrupts temporarily until it has correctly saved state and jumped to the ISR routine. It is the responsibility of the ISR to re-enable the interrupts. To prevent the *IntSource* register indicating the incorrect source for an interrupt level, the ISR must read and store the *IntSource* value before re-enabling the interrupts via the Enable Traps (ET) field in the Processor State Register (PSR) of the LEON.

See section 11.9 on page 104 for a complete description of the interrupt handling procedure.

After reset the state machine remains in *Idle* state until an interrupt acknowledge is received from the CPU (indicated by *cpu_iack*). When the acknowledge is received the state machine transitions to the *Compare* state, resetting the source counter (*cnt*) to the number of interrupt sources.

While in the *Compare* state the state machine cycles through each possible interrupt source in decrementing order. For each active interrupt source the programmed priority (*int_priority[cnt][3:0]*) is compared with the acknowledged interrupt level from the CPU (*cpu_icu_ilevel*), if they match then the interrupt is considered the new winner. This implies the last interrupt source checked has the highest priority, e.g interrupt source zero has the highest priority and the first source checked has the lowest priority. After all interrupt sources are checked the state machine transitions to the *IntClear* state, and updates the *int_source* register on the transition.

Should there be no active interrupts for the acknowledged level (e.g. a level sensitive interrupt was removed), the *IntSource* register will be set to *NoInterrupt*. *NoInterrupt* is defined as the highest possible value that *IntSource* can be set to (in this case 0x1F), and the state machine will return to *Idle*.

The exact number of compares performed per clock cycle is dependent the number of interrupts, and logic area to logic speed trade-off, and is left to the implementer to determine. A comparison of

all interrupt sources must complete within 8 clock cycles (determined by the CPU acknowledge hardware).

When in the *IntClear* state the state machine has determined the interrupt source to clear (indicated by the *int_source* register). It resets the pending bit for that interrupt source, transitions back to the *Idle* state and waits for the next acknowledge from the CPU.

The minimum time between successive interrupt acknowledges from the CPU is 8 cycles.

15 Timers Block (TIM)

The Timers block contains general purpose timers, a watchdog timer and timing pulse generator for use in other sections of SoPEC.

15.1 WATCHDOG TIMER

The watchdog timer is a 32 bit counter value which counts down each time a timing pulse is received. The period of the timing pulse is selected by the *WatchDogUnitSel* register. The value at any time can be read from the *WatchDogTimer* register and the counter can be reset by writing a non-zero value to the register. When the counter transitions from 1 to 0, a system wide reset will be triggered as if the reset came from a hardware pin.

The watchdog timer can be polled by the CPU and reset each time it gets close to 1, or alternatively a threshold (*WatchDogIntThres*) can be set to trigger an interrupt for the watchdog timer to be serviced by the CPU. If the *WatchDogIntThres* is set to N, then the interrupt will be triggered on the N to N-1 transition of the *WatchDogTimer*. This interrupt can be effectively masked by setting the threshold to zero. The watchdog timer can be disabled, without causing a reset, by writing zero to the *WatchDogTimer* register.

15.2 TIMING PULSE GENERATOR

The timing block contains a timing pulse generator clocked by the system clock, used to generate timing pulses of programmable periods. The period is programmed by accessing the *TimerStartValue* registers. Each pulse is of one system clock duration and is active high, with the pulse period accurate to the system clock frequency. The periods after reset are set to 1us, 100us and 100 ms.

The timing pulse generator also contains a 64-bit free running counter that can be read or reset by accessing the *FreeRunCount* registers. The free running counter can be used to determine elapsed time between events at system clock accuracy or could be used as an input source in low-security random number generator.

15.3 GENERIC TIMERS

SoPEC contains 3 programmable generic timing counters, for use by the CPU to time the system. The timers are programmed to a particular value and count down each time a timing pulse is received. When a particular timer decrements from 1 to 0, an interrupt is generated. The counter can be programmed to automatically restart the count, or wait until re-programmed by the CPU. At any time the status of the counter can be read from *GenCntValue*, or can be reset by writing to *GenCntValue* register. The auto-restart is activated by setting the *GenCntAuto* register, when activated the counter restarts at *GenCntStartValue*. A counter can be stopped or started at any

time, without affecting the contents of the *GenCntValue* register, by writing a 1 or 0 to the relevent *GenCntEnable* register.

15.4   IMPLEMENTATION

15.4.1   Definitions of I/O

5

Table 91. Timers block I/O definition

Port name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
tim_pulse[2:0]	3	Out	Timers block generated timing pulses, each one *pclk* wide 0 - Nominal 1µs pulse 1 - Nominal 100 µs pulse 2 - Nominal 10ms pulse
CPU interface			
cpu_adr[6:2]	5	In	CPU address bus. Only 5 bits are required to decode the address space for the ICU block
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
tim_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_tim_sel	1	In	Block select from the CPU. When *cpu_tim_sel* is high both *cpu_adr* and *cpu_dataout* are valid
tim_cpu_rdy	1	Out	Ready signal to the CPU. When *tim_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the TIM block and for a read cycle this means the data on *tim_cpu_data* is valid.
tim_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
tim_cpu_debug_valid	1	Out	Debug Data valid on *tim_cpu_data* bus. Active high
Miscellaneous			
tim_icu_wd_irq	1	Out	Watchdog timer interrupt signal to the ICU block
tim_icu_irq[2:0]	3	Out	Generic timer interrupt signals to the ICU block

tim_cpr_reset_n	1	Out	Watch dog timer system reset.

15.4.2 Timers sub-block partition

15.4.3 Watchdog timer

The watchdog timer counts down from pre-programmed value, and generates a system wide reset when equal to one. When the counter passes a pre-programmed threshold (*wdog_tim_thres*) value an interrupt is generated (*tim_icu_wd_irq*) requesting the CPU to update the counter. Setting the counter to zero disables the watchdog reset. In supervisor mode the watchdog counter can be written to or read from at any time, in user mode access is denied. Any accesses in user mode will generate a bus error.

The counter logic is given by

```
if (wdog_wen == 1) then
   wdog_tim_cnt = write_data        // load new data
elsif ( wdog_tim_cnt == 0) then
   wdog_tim_cnt = wdog_tim_cnt      // count disabled
elsif ( cnt_en == 1 ) then
   wdog_tim_cnt--
else
   wdog_tim_cnt = wdog_tim_cnt
```

The timer decode logic is

```
if (( wdog_tim_cnt == wdog_tim_thres) AND (wdog_tim_cnt != 0
) AND (cnt_en == 1)) then
   tim_icu_wd_irq = 1
else
   tim_icu_wd_irq = 0
// reset generator logic
if (wdog_tim_cnt == 1) AND (cnt_en == 1) then
   tim_cpr_reset_n = 0
else
   tim_cpr_reset_n = 1
```

15.4.4 Generic timers

The generic timers block consists of 3 identical counters. A timer is set to a pre-configured value (*GenCntStartValue*) and counts down once per selected timing pulse (*gen_unit_sel*). The timer can be enabled or disabled at any time (*gen_tim_en*), when disabled the counter is stopped but not cleared. The timer can be set to automatically restart (*gen_tim_auto*) after it generates an interrupt. In supervisor mode a timer can be written to or read from at any time, in user mode access is determined by the *GenCntUserModeEnable* register settings.

The counter logic is given by

```
if (gen_wen == 1) then
   gen_tim_cnt = write_data
elsif (( cnt_en == 1 )AND (gen_tim_en == 1 )) then
```

```
                if ( gen_tim_cnt == 1) OR ( gen_tim_cnt == 0)    then  //
            counter may need re-starting
                if (gen_tim_auto == 1) then
                    gen_tim_cnt = gen_tim_cnt_st_value
                else
                    gen_tim_cnt = 0                                    // hold
            count at zero
                else
                    gen_tim_cnt--
            else
                gen_tim_cnt = gen_tim_cnt
```

The decode logic is

```
            if (gen_tim_cnt == 1)AND ( cnt_en == 1 )AND (gen_tim_en == 1
            ) then
                tim_icu_irq = 1
            else
                tim_icu_irq = 0
```

15.4.5   Timing pulse generator

The timing pulse generator contains a general free running 64-bit timer and 3 timing pulse generators producing timing pulses of one cycle duration with a programmable period. The period is programmed by changed the *TimerStartValue* registers, but have a nominal starting period of 1µs, 100µs and 1ms. In supervisor mode the free running timer register can be written to or read from at any time, in user mode access is denied. The status of each of the timers can be read by accessing the *PulseTimerStatus* registers in supervisor mode. Any accesses in user mode will result in a bus error.

15.4.5.1 Free Run Timer

The increment logic block increments the timer count on each clock cycle. The counter wraps around to zero and continues incrementing if overflow occurs. When the timing register (*FreeRunCount*) is written to, the configuration registers block will set the *free_run_wen* high for a clock cycle and the value on *write_data* will become the new count value. If *free_run_wen[1]* is 1 the higher 32 bits of the counter will be written to, otherwise if *free_run_wen[0]* the lower 32 bits are written to. It is the responsibility of software to handle these writes in a sensible manner.

The increment logic is given by

```
            if (free_run_wen[1] == 1) then
                free_run_cnt[63:32] = write_data
            elsif (free_run_wen[0] == 1) then
                free_run_cnt[31:0]  = write_data
            else
                free_run_cnt ++
```

15.4.5.2 Pulse Timers

The pulse timer logic generates timing pulses of 1 clock cycle length and programmable period. Nominally they generate pulse periods of 1µs, 100µs and 1ms. The logic for timer 0 is given by:

```
            // Nominal 1us generator
```

```
            if (pulse_0_cnt == 0 ) then
               pulse_0_cnt = timer_start_value[0]
               tim_pulse[0] = 1
            else
               pulse_0_cnt --
               tim_pulse[0] = 0
```

The logic for timer 1 is given by:

```
            // 100us generator
            if ((pulse_1_cnt == 0) AND (tim_pulse[0] == 1)) then
               pulse_1_cnt = timer_start_value[1]  .
               tim_pulse[1] = 1
            elsif (tim_pulse[0] == 1) then
               pulse_1_cnt --
               tim_pulse[1] = 0
            else
               pulse_1_cnt = pulse_1_cnt
               tim_pulse[1] = 0
```

The logic for the timer 2 is given by:

```
            // 10ms generator
            if ((pulse_2_cnt == 0 ) AND (tim_pulse[1] == 1)) then
               pulse_2_cnt = timer_start_value[2]
               tim_pulse[2] = 1
            elsif (tim_pulse[1] == 1) then
               pulse_2_cnt --
               tim_pulse[2] = 0
            else
               pulse_2_cnt = pulse_2_cnt
               tim_pulse[2] = 0
```

15.4.6   Configuration registers

The configuration registers in the TIM are programmed via the CPU interface. Refer to section 11.4.3 on page 69 for a description of the protocol and timing diagrams for reading and writing registers in the TIM. Note that since addresses in SoPEC are byte aligned and the CPU only supports 32-bit register reads and writes, the lower 2 bits of the CPU address bus are not required to decode the address space for the TIM. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *tim_pcu_data*. Table 92 lists the configuration registers in the TIM block .

Table 92. Timers Register Map

Address TIM_base +	Register	#bits	Reset	Description
0x00	WatchDogUnitSel	2	0x0	Specifies the units used for the

200

				watchdog timer: 0 - Nominal 1 μs pulse 1 - Nominal 100 μs pulse 2 - Nominal 10 ms pulse 3 - *pclk*
0x04	WatchDogTimer	32	0xFFFF _FFFF	Specifies the number of units to count before watchdog timer triggers.
0x08	WatchDogIntThres	32	0x0000 _0000	Specifies the threshold value below which the watchdog timer issues an interrupt
0x0C-0x10	FreeRunCount[1:0]	2x32	0x0000 _0000	Direct access to the free running counter register. Bus 0 - Access to bits 31-0 Bus 1 - Access to bits 63-32
0x14 to 0x1C	GenCntStartValue[2:0]	3x32	0x0000 _0000	Generic timer counter start value, number of units to count before event
0x20 to 0x28	GenCntValue[2:0]	3x32	0x0000 _0000	Direct access to generic timer counter registers
0x2C to 0x34	GenCntUnitSel[2:0]	3x2	0x0	Generic counter unit select. Selects the timing units used with corresponding counter: 0 - Nominal1 μs pulse 1 - Nominal100 μs pulse 2 - Nominal 10 ms pulse 3 - *pclk*
0x38 to 0x40	GenCntAuto[2:0]	3x1	0x0	Generic counter auto re-start select. When high timer automatically restarts, otherwise timer stops.
0x44 to 0x4C	GenCntEnable[2:0]	3x1	0x0	Generic counter enable. 0 - Counter disabled 1 - Counter enabled
0x50	GenCntUserMode Enable	3	0x0	User Mode Access enable to generic timer configuration register. When 1 user access is enabled. Bit 0 - Generic timer 0 Bit 1 - Generic timer 1 Bit 2 - Generic timer 2
0x54 to 0x5C	TimerStartValue[2:0]	3x8	0x7F, 0x63,	Timing pulse generator start value. Indicates the start value for each

			0x63	timing pulse timers. For timer 0 the start value specifies the timer period in *pclk* cycles - 1. For timer 1 the start value specifies the timer period in timer 0 intervals -1 For timer 2 the start value specifies the timer period in timer 1 intervals - 1. Nominally the timers generate pulses at 1us,100us and 10ms intervals respecitively.
0x60	DebugSelect[6:2]	5	0x00	Debug address select. Indicates the address of the register to report on the *tim_cpu_data* bus when it is not otherwise being used.
Read Only Registers				
0x64	PulseTimerStatus	24	0x00	Current pulse timer values, and pulses 7:0 - Timer 0 count 15:8 - Timer 1 count 23:16 - Timer 2 count 24 - Timer 0 pulse 25 - Timer 1 pulse 26 - Timer 2 pulse

15.4.6.1 Supervisor and user mode access

The configuration registers block examines the CPU access type (*cpu_acode* signal) and determines if the access is allowed to that particular register, based on configured user access registers. If an access is not allowed the block will issue a bus error by asserting the *tim_cpu_berr* signal.

The timers block is fully accessible in supervisor data mode, all registers can written to and read from. In user mode access is denied to all registers in the block except for the generic timer configuration registers that are granted user data access. User data access for a generic timer is granted by setting corresponding bit in the *GenCntUserModeEnable* register. This can only be changed in supervisor data mode. If a particular timer is granted user data access then all registers for configuring that timer will be accessible. For example if timer 0 is granted user data access the *GenCntStartValue[0]*, *GenCntUnitSel[0]*, *GenCntAuto[0]*, *GenCntEnable[0]* and *GenCntValue[0]* registers can all be written to and read from without any restriction.

Attempts to access a user data mode disabled timer configuration register will result in a bus error.

Table 93 details the access modes allowed for registers in the TIM block. In supervisor data mode all registers are accessable. All forbidden accesses will result in a bus error (*tim_cpu_berr* asserted).

Table 93. TIM supervisor and user access modes

Register Address	Registers	Access Permission
0x00	WatchDogUnitSel	Supervisor data mode only
0x04	WatchDogTimer	Supervisor data mode only
0x08	WatchDogIntThres	Supervisor data mode only
0x0C-0x10	FreeRunCount	Supervisor data mode only
0x14	GenCntStartValue[0]	GenCntUserModeEnable[0]
0x18	GenCntStartValue[1]	GenCntUserModeEnable[1]
0x1C	GenCntStartValue[2]	GenCntUserModeEnable[2]
0x20	GenCntValue[0]	GenCntUserModeEnable[0]
0x24	GenCntValue[1]	GenCntUserModeEnable[1]
0x28	GenCntValue[2]	GenCntUserModeEnable[2]
0x2C	GenCntUnitSel[0]	GenCntUserModeEnable[0]
0x30	GenCntUnitSel[1]	GenCntUserModeEnable[1]
0x34	GenCntUnitSel[2]	GenCntUserModeEnable[2]
0x38	GenCntAuto[0]	GenCntUserModeEnable[0]
0x3C	GenCntAuto[1]	GenCntUserModeEnable[1]
0x40	GenCntAuto[2]	GenCntUserModeEnable[2]
0x44	GenCntEnable[0]	GenCntUserModeEnable[0]
0x48	GenCntEnable[1]	GenCntUserModeEnable[1]
0x4C	GenCntEnable[2]	GenCntUserModeEnable[2]
0x50	GenCntUserModeEnable	Supervisor data mode only
0x54-0x5C	TimerStartValue[2:0]	Supervisor data mode only
0x60	DebugSelect	Supervisor data mode only
0x64	PulseTimerStatus	Supervisor data mode only

16 Clocking, Power and Reset (CPR)

The CPR block provides all of the clock, power enable and reset signals to the SoPEC device.

16.1 POWERDOWN MODES

The CPR block is capable of powering down certain sections of the SoPEC device. When a section is powered down (i.e. put in sleep mode) no state is retained(except the PSS storage), the CPU must re-initialize the section before it can be used again.

For the purpose of powerdown the SoPEC device is divided into sections:

Table 94. Powerdown sectioning

Section	Block
Print Engine Pipeline SubSystem (Section 0)	PCU
	CDU
	CFU
	LBD
	SFU
	TE
	TFU
	HCU
	DNC
	DWU
	LLU
	PHI
CPU-DRAM (Section 1)	DRAM
	CPU/MMU
	DIU
	TIM
	ROM
	LSS
	PSS
	ICU
ISI Subsystem (Section 2)	ISI (SCB)
	DMA Ctrl (SCB)
	GPIO
USB Subsystem (Section 3)	USB (SCB)

Note that the CPR block is not located in any section. All configuration registers in the CPR block are clocked by an ungateable clock and have special reset conditions.

16.1.1   Sleep mode

Each section can be put into sleep mode by setting the corresponding bit in the *SleepModeEnable* register. To re-enable the section the sleep mode bit needs to be cleared and then the section should be reset by writing to the relevant bit in the *ResetSection* register. Each block within the section should then be re-configured by the CPU.

If the CPU system (section 1) is put into sleep mode, the SoPEC device will remain in sleep mode until a system level reset is initiated from the reset pin, or a wakeup reset by the SCB block as a result of activity on either the USB or ISI bus. The watchdog timer cannot reset the device as it is in section 1 also, and will be in sleep mode.

If the CPU and ISI subsystem are in sleep mode only a reset from the USB or a hardware reset will re-activate the SoPEC device.

204

If all sections are put into sleep mode, then only a system level reset initiated by the reset pin will re-activate the SoPEC device.

Like all software resets in SoPEC the *ResetSection* register is active-low i.e. a 0 should be written to each bit position requiring a reset. The *ResetSection* register is self-reseting.

5   16.1.2   Sleep Mode powerdown procedure

When powering down a section, the section may retain it's current state (although not gauranteed to). It is possible when powering back up a section that inconsistancies between interface state machines could cause incorrect operation. In order to prevent such condition from happening, all blocks in a section must be disabled before powering down. This will ensure that blocks are

10   restored in a benign state when powered back up.

In the case of PEP section units setting the *Go* bit to zero will disable the block. The DRAM subsystem can be effectively disabled by setting the *RotationSync* bit to zero, and the SCB system disabled by setting the *DMAAccessEn* bits to zero turning off the DMA access to DRAM. Other CPU subsystem blocks without any DRAM access do not need to be disabled.

15   16.2   RESET SOURCE

The SoPEC device can be reset by a number of sources. When a reset from an internal source is initiated the reset source register (*ResetSrc*) stores the reset source value. This register can then be used by the CPU to determine the type of boot sequence required.

16.3   CLOCK RELATIONSHIP

20   The crystal oscillator excites a 32MHz crystal through the xtalin and xtalout pins. The 32MHz output is used by the PLL to derive the master VCO frequency of 960MHz. The master clock is then divided to produce 320MHz clock (*clk320*), 160MHz clock (*clk160*) and 48MHz (*clk48*) clock sources.

The phase relationship of each clock from the PLL will be defined. The relationship of internal

25   clocks *clk320, clk48* and *clk160* to *xtalin* will be undefined.

At the output of the clock block, the skew between each *pclk* domain (*pclk_section[2:0]* and *jclk*) should be within skew tolerances of their respective domains (defined as less than the hold time of a D-type flip flop).

The skew between *doclk* and *pclk* should also be less than the skew tolerances of their respective

30   domains.

The *usbclk* is derived from the PLL output and has no relationship with the other clocks in the system and is considered asynchronous.

16.4   PLL CONTROL

The PLL in SoPEC can be adjusted by programming the *PLLRangeA, PLLRangeB, PLLTunebits*

35   and *PLLMult* registers. If these registers are changed by the CPU the values are not updated until the *PLLUpdate* register is written to. Writing to the *PLLUpdate* register triggers the PLL control state machine to update the PLL configuration in a safe way. When an update is active (as indicated by *PLLUpdate* register) the CPU must not change any of the configuration registers, doing so could cause the PLL to lose lock indefintely, requiring a hardware reset to recover. Configuring the PLL

registers in an inconsistent way can also cause the PLL to lose lock, care must taken to keep the PLL configuration within specified parameters.

The VCO frequency of the PLL is calculated by the number of divider in the feedback path. PLL output A is used as the feedback source.

VCOfreq = *REFCLK* x *PLLMult* x *PLLRangeA* x External divider

VCOfreq = 32   x 3 x 10 x 1 = 960 Mhz.

In the default PLL setup, *PLLMult* is set to 3, *PLLRangeA* is set to 3 which corresponds to a divide by 10, *PLLRangeB* is set to 5 which corresponds to a divide by 3.

PLLouta = VCOfreq / *PLLRangeA* = 960Mhz / 10 = 96 Mhz

PLLoutb = VCOfreq / *PLLRangeB* = 960Mhz / 3 = 320 Mhz

See [16] for complete PLL setup parameters.

16.5   IMPLEMENTATION

16.5.1   Definitions of I/O

Table 95. CPR I/O definition

Port name	Pins	I/O	Description
Clocks and Resets			
Xtalin	1	In	Crystal input, direct from IO pin.
Xtalout	1	Inout	Crystal output, direct to IO pin.
pclk_section[3:0]	4	Out	System clocks for each section
Doclk	1	Out	Data out clock (2x *pclk*) for the PHI block
Jclk	1	Out	Gated version of system clock used to clock the JPEG decoder core in the CDU
Usbclk	1	Out	USB clock, nominally at 48 Mhz
jclk_enable	1	In	Gating signal for *jclk*. When 1 *jclk* is enabled
reset_n	1	In	Reset signal from the *reset_n* pin
usb_cpr_reset_n	1	In	Reset signal from the USB block
isi_cpr_reset_n	1	In	Reset signal from the ISI block
tim_cpr_reset_n	1	In	Reset signal from watch dog timer.
gpio_cpr_wakeup	1	In	SoPEC wake up from the GPIO, active high.
prst_n_section[3:0]	4	Out	System resets for each section, synchronous active low
dorst_n	1	Out	Reset for PHI block, synchronous to *doclk*
jrst_n	1	Out	Reset for JPEG decoder core in CDU block, synchronous to *jclk*
usbrst_n	1	Out	Reset for the USB block, synchronous to *usbclk*
CPU interface			
cpu_adr[5:2]	3	In	CPU address bus. Only 4 bits are required to decode the address space for the CPR block

cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
cpr_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_cpr_sel	1	In	Block select from the CPU. When *cpu_cpr_sel* is high both *cpu_adr* and *cpu_dataout* are valid
cpr_cpu_rdy	1	Out	Ready signal to the CPU. When *cpr_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the block and for a read cycle this means the data on *cpr_cpu_data* is valid.
cpr_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
cpr_cpu_debug_valid	1	Out	Debug Data valid on *cpr_cpu_data* bus. Active high

16.5.2   Configuration registers

The configuration registers in the CPR are programmed via the CPU interface. Refer to section 11.4 on page 69 for a description of the protocol and timing diagrams for reading and writing registers in the CPR. Note that since addresses in SoPEC are byte aligned and the CPU only supports 32-bit register reads and writes, the lower 2 bits of the CPU address bus are not required to decode the address space for the CPR. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *cpr_pcu_data*. Table 96 lists the configuration registers in the CPR block.

The CPR block will only allow supervisor data mode accesses (i.e. *cpu_acode[1:0]* = *SUPERVISOR_DATA*). All other accesses will result in *cpr_cpu_berr* being asserted .

Table 96. CPR Register Map

Address CPR_base +	Register	#bits	Reset	Description
0x00	SleepModeEnable	4	0x0[a]	Sleep Mode enable, when high a section of logic is put into powerdown. Bit 0 - Controls section 0 Bit 1 - Controls section 1 Bit 2 - Controls section 2

				Bit 3 - Controls section 3 Note that the SleepModeEnable register has special reset conditions. See Section 16.5.6 for details
0x04	ResetSrc	5	0x1[a]	Reset Source register, indicating the source of the last reset (or wake-up) Bit 0 - External Reset Bit 1 - USB wakeup reset Bit 2 - ISI wakeup reset Bit 3 - Watchdog timer reset Bit 4 - GPIO wake-up (Read Only Register)
0x08	ResetSection	4	0xF	Active-low synchronous reset for each section, self-resetting. Bit 0 - Controls section 0 Bit 1 - Controls section 1 Bit 2 - Controls section 2 Bit 3 - Controls section 3
0x0C	DebugSelect[5:2]	4	0x0	Debug address select. Indicates the address of the register to report on the cpr_cpu_data bus when it is not otherwise being used.
PLL Control				
0x10	PLLTuneBits	10	0x3BC	PLL tuning bits
0x14	PLLRangeA	4	0x3	PLLOUT A frequency selector (defaults to 60Mhz to 125Mhz)
0x18	PLLRangeB	3	0x5	PLLOUT B frequency selector (defaults to 200Mhz to 400Mhz)
0x1C	PLLMultiplier	5	0x03	PLL multiplier selector, defaults to refclk x 3
0x20	PLLUpdate	1	0x0	PLL update control. A write (of any value) to this register will cause the PLL to lose lock for ~100us. Reading the register indicates the status of the update. 0 - PLL update complete 1 - PLL update active No writes to PLLTuneBits,PLLRangeA,PLL-

				RangeB,PLLMultiplier or *PLLUpdate* are allowed while the PLL update is active.

a.   Reset value depends on reset source. External reset shown.

16.5.3   CPR Sub-block partition

16.5.4   reset_n deglitch

The external reset_n signal is deglitched for about 1µs. reset_n must maintain a state for 1us second before the state is passed into the rest of the device. All deglitch logic is clocked on *bufrefclk*.

16.5.5   Sync reset

The reset synchronizer retimes an asynchronous reset signal to the clock domain that it resets. The circuit prevents the inactive edge of reset occurring when the clock is rising

16.5.6   Reset generator logic

The reset generator logic is used to determine which clock domains should be reset, based on configured reset values (*reset_section_n*), the external reset (*reset_n*), watchdog timer reset (*tim_cpr_reset_n*), the USB reset (*usb_cpr_reset_n*), the GPIO wakeup control (*gpio_cpr_wakeup*) and the ISI reset (*isi_cpr_reset_n*). The reset direct from the IO pin (*reset_n*) is synchronized and de-glitched before feeding the reset logic.

All resets are lengthened to at least 16 *pclk* cycles, regardless of the duration of the input reset. The clock for a particular section must be running for the reset to have an effect. The clocks to each section can be enabled/disabled using the *SleepModeEnable* register.

Resets from the ISI or USB block reset everything except its own section (section 2 or 3).

Table 97. Reset domains

Reset signal	Domain
reset_dom[0]	Section 0 pclk domain (PEP)
reset_dom[1]	Section 1 pclk domain (CPU)
reset_dom[2]	Section 2 pclk domain (ISI)
reset_dom[3]	Section 3 usbclk/pclk domain (USB)
reset_dom[4]	doclk domain
reset_dom[5]	jclk domain

The logic is given by

```
if (reset_dg_n == 0) then
    reset_dom[5:0]      = 0x00      // reset everything
    reset_src[4:0]      = 0x01
    cfg_reset_n         = 0
    sleep_mode_en[3:0]  = 0x0       // re-awaken all sections
elsif (tim_cpr_reset_n == 0) then
```

209

```
          reset_dom[5:0]        = 0x00        // reset everything except
      CPR config
          reset_src[4:0]        = 0x08
          cfg_reset_n           = 1           // CPR config stays the same
          sleep_mode_en[1]      = 0            // re-awaken section 1 only
      (awake already)
      elsif (usb_cpr_reset_n == 0) then
          reset_dom[5:0]        = 0x08         // all except USB domain +
      CPR config
          reset_src[4:0]        = 0x02
          cfg_reset_n           = 1           // CPR config stays the same
          sleep_mode_en[1]      = 0            // re-awaken section 1 only,
      section 3 is awake
      elsif (isi_cpr_reset_n == 0) then
          reset_dom[5:0]        = 0x04         // all except ISI domain +
      CPR config
          reset_src[4:0]        = 0x04
          cfg_reset_n           = 1           // CPR config stays the same
          sleep_mode_en[1]      = 0            // re-awaken section 1 only,
      section 2 is awake
      elsif (gpio_cpr_wakeup = 1) then
          reset_dom[5:0]        = 0x3C         // PEP and CPU sections only
          reset_src[4:0]        = 0x10
          cfg_reset_n           = 1           // CPR config stays the same
          sleep_mode_en[1]      = 0            // re-awaken section 1 only,
      section 2 is awake
      else
          // propagate resets from reset section register
          reset_dom[5:0]     .   = 0x3F                   // default to on
          cfg_reset_n            = 1                          // CPR cfg
      registers are not in any section
          sleep_mode_en[3:0]    = sleep_mode_en[3:0]  // stay the same
      by default
          if (reset_section_n[0] == 0) then
              reset_dom[5] = 0                   // jclk domain
              reset_dom[4] = 0                   // doclk domain
              reset_dom[0] = 0                   // pclk section 0 domain
          if (reset_section_n[1] == 0) then
              reset_dom[1] = 0                   // pclk section 1 domain
          if (reset_section_n[2] == 0) then
              reset_dom[2] = 0                    // pclk section 2 domain
      (ISI)
          if (reset_section_n[3] == 0) then
              reset_dom[3] = 0                   // USB domain
```

16.5.7   Sleep logic

The sleep logic is used to generate gating signals for each of SoPECs clock domains. The gate enable (*gate_dom*) is generated based on the configured *sleep_mode_en* and the internally generated *jclk_enable* signal.

The logic is given by

```
// clock gating for sleep modes
gate_dom[5:0] = 0x0                  // default to all clocks
on
if (sleep_mode_en[0] == 1) then   // section 0 sleep
   gate_dom[0] = 1                 // pclk section 0
   gate_dom[4] = 1                 // doclk domain
   gate_dom[5] = 1                 // jclk domain
if (sleep_mode_en[1] == 1) then   // section 1 sleep
   gate_dom[1] = 1                 // pclk section 1
if (sleep_mode_en[2] == 1) then   // section 2 sleep
   gate_dom[2] = 1                 // pclk section 2
if (sleep_mode_en[3] == 1) then   // section 3 sleep
   gate_dom[3] = 1                 // usb section 3
// the jclk can be turned off by CDU signal
if (jclk_enable == 0) then
   gate_dom[5] = 1
```

The clock gating and sleep logic is clocked with the *master_pclk* clock which is not gated by this logic, but is synchronous to other *pclk_section* and *jclk* domains.

Once a section is in sleep mode it cannot generate a reset to restart the device. For example if section 1 is in sleep mode then the watchdog timer is effectively disabled and cannot trigger a reset.

16.5.8   Clock gate logic

The clock gate logic is used to safely gate clocks without generating any glitches on the gated clock. When the enable is high the clock is active otherwise the clock is gated.

16.5.9   Clock generator Logic

The clock generator block contains the PLL, crystal oscillator, clock dividers and associated control logic. The PLL VCO frequency is at 960MHz locked to a 32 MHz *refclk* generated by the crystal oscillator. In test mode the *xtalin* signal can be driven directly by the test clock generator, the test clock will be reflected on the *refclk* signal to the PLL.

16.5.9.1 Clock divider A

The clock divider A block generates the 48MHz clock from the input 96MHz clock (*pllouta*) generated by the PLL. The divider is enabled only when the PLL has acquired lock.

16.5.9.2 Clock divider B

The clock divider B block generates the 160MHz clocks from the input 320MHz clock (*plloutb*) generated by the PLL. The divider is enabled only when the PLL has acquired lock.

16.5.9.3 PLL control state machine

The PLL will go out of lock whenever *pll_reset* goes high (the PLL reset is the only active high reset in the device) or if the configuration bits *pll_rangea*, *pll_rangeb*, *pll_mult*, *pll_tune* are changed. The

PLL control state machine ensures that the rest of the device is protected from glitching clocks while the PLL is being reset or it's configuration is being changed.

In the case of a hardware reset (the reset is deglitched), the state machine first disables the output clocks (via the *clk_gate* signal), it then holds the PLL in reset while its configuration bits are reset to default values. The state machine then releases the PLL reset and waits approx. 100us to allow the PLL to regain lock. Once the lock time has elapsed the state machine re-enables the output clocks and resets the remainder of the device via the *reset_dg_n* signal.

When the CPU changes any of the configuration registers it must write to the PLLupdate register to allow the state machine to update the PLL to the new configuration setup. If a PLLUpdate is detected the state machine first gates the output clocks. It then holds the PLL in reset while the PLL configuration registers are updated. Once updated the PLL reset is released and the state machine waits approx 100us for the PLL to regain lock before re-enabling the output clocks. Any write to the PLLUpdate register will cause the state machine to perform the update operation regardless of whether the configuration values changed or not.

All logic in the clock generator is clocked on *bufrefclk* which is always an active clock regardless of the state of the PLL.

17 ROM Block

17.1 OVERVIEW

The ROM block interfaces to the CPU bus and contains the SoPEC boot code. The ROM block consists of the CPU bus interface, the ROM macro and the ChipID macro. The current ROM size is 16 KBytes implemented as a 4096 x32 macro. Access to the ROM is not cached because the CPU enjoys fast (no more than one cycle slower than a cache access), unarbitrated access to the ROM. Each SoPEC device is required to have a unique ChipID which is set by blowing fuses at manufacture. IBM's 300mm ECID macro and a custom 112-bit ECID macro are used to implement the ChipID offering 224-bits of laser fuses. The exact number of fuse bits to be used for the ChipID will be determined later but all bits are made available to the CPU. The ECID macros allows all 224 bits to be read out in parallel and the ROM block will make all 224 bits available in the *FuseChipID[N]* registers which are readable by the CPU in supervisor mode only.

17.2 BOOT OPERATION

The are two boot scenarios for the SoPEC device namely after power-on and after being awoken from sleep mode. When the device is in sleep mode it is hoped that power will actually be removed from the DRAM, CPU and most other peripherals and so the program code will need to be freshly downloaded each time the device wakes up from sleep mode. In order to reduce the wakeup boot time (and hence the perceived print latency) certain data items are stored in the PSS block (see section 18). These data items include the SHA-1 hash digest expected for the program(s) to be downloaded, the master/slave SoPEC id and some configuration parameters. All of these data items are stored in the PSS by the CPU prior to entering sleep mode. The SHA-1 value stored in the PSS is calculated by the CPU by decrypting the signature of the downloaded program using the appropriate public key stored in ROM. This compute intensive decryption only needs to take place once as part of the power-on boot sequence - subsequent wakeup boot sequences will simply use

the resulting SHA-1 digest stored in the PSS. Note that the digest only needs to be stored in the PSS before entering sleep mode and the PSS can be used for temporary storage of any data at all other times.

The CPU is expected to be in supervisor mode for the entire boot sequence described by the pseudocode below. Note that the boot sequence has not been finalised but is expected to be close to the following:

```
if (ResetSrc == 1)   then   // Reset was a power-on reset
    configure_sopec   // need to configure peris (USB, ISI,
DMA, ICU etc.)
    // Otherwise reset was a wakeup reset so peris etc. were
already configured
    PAUSE: wait until IrqSemaphore != 0   // i.e. wait until an
interrupt has been serviced
    if (IrqSemaphore == DMAChan0Msg) then
        parse_msg(DMAChan0MsgPtr)   // this routine will parse the
message and take any
                              // necessary action e.g. programming
the DMAChannel1 registers
    elsif (IrqSemaphore == DMAChan1Msg) then   // program has
been downloaded
        CalculatedHash = gen_sha1(ProgramLocn, ProgramSize)
        if (ResetSrc == 1)   then
            ExpectedHash = sig_decrypt(ProgramSig,public_key)
        else
            ExpectedHash = PSSHash
        if (ExpectedHash == CalculatedHash) then
            jmp(PrgramLocn)   // transfer control to the downloaded
program
        else
            send_host_msg("Program Authentication Failed")
            goto PAUSE:
    elsif (IrqSemaphore == timeout) then   // nothing has
happened
        if (ResetSrc == 1)   then

            sleep_mode()   // put SoPEC into sleep mode to be woken
up by USB/ISI activity
        else   // we were woken up but nothing happened
            reset_sopec(PowerOnReset)
    else
        goto PAUSE
```

The boot code places no restrictions on the activity of any programs downloaded and authenticated by it other than those imposed by the configuration of the MMU i.e. the principal function of the boot

code is to authenticate that any programs downloaded by it are from a trusted source. It is the responsibility of the downloaded program to ensure that any code it downloads is also authenticated and that the system remains secure. The downloaded program code is also responsible for setting the SoPEC ISIId (see section 12.5 for a description of the ISIId) in a multi -SoPEC system. See the "SoPEC Security Overview" document [9] for more details of the SoPEC security features.

17.3   IMPLEMENTATION

17.3.1   Definitions of I/O

Table 98. ROM Block I/O

Port name	Pins	I/O	Description
Clocks and Resets			
prst_n	1	In	Global reset. Synchronous to pclk, active low.
Pclk	1	In	Global clock
CPU Interface			
cpu_adr[14:2]	13	In	CPU address bus. Only 13 bits are required to decode the address space for this block.
rom_cpu_data[31: 0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
cpu_rom_sel	1	In	Block select from the CPU. When *cpu_rom_sel* is high *cpu_adr* is valid
rom_cpu_rdy	1	Out	Ready signal to the CPU. When *rom_cpu_rdy* is high it indicates the last cycle of the access. For a read cycle this means the data on *rom_cpu_data* is valid.
rom_cpu_berr	1	Out	ROM bus error signal to the CPU indicating an invalid access.

17.3.2   Configuration registers

The ROM block will only allow read accesses to the *FuseChipID* registers and the ROM with supervisor data space permissions (i.e. *cpu_acode[1:0]* = 11). Write accesses with supervisor data space permissions

will have no effect. All other accesses with will result in *rom_cpu_berr* being asserted. The CPU subsystem bus slave interface is described in more detail in section 9.4.3.

Table 99. ROM Block Register Map

214

Address ROM_base +	Register	#bits	Reset	Description
0x4000	FuseChipID0	32	n/a	Value of corresponding fuse bits 31 to 0 of the IBM 112-bit ECID macro. (Read only)
0x4004	FuseChipID1	32	n/a	Value of corresponding fuse bits 63 to 32 of the IBM 112-bit ECID macro. (Read only)
0x4008	FuseChipID2	32	n/a	Value of corresponding fuse bits 95 to 64 of the IBM 112-bit ECID macro. (Read only)
0x400C	FuseChipID3	16	n/a	Value of corresponding fuse bits 111 to 96 of the IBM 112-bit ECID macro. (Read only)
0x4010	FuseChipID4	32	n/a	Value of corresponding fuse bits 31 to 0 of the Custom 112-bit ECID macro. (Read only)
0x4014	FuseChipID5	32	n/a	Value of corresponding fuse bits 63 to 32 of the Custom 112-bit ECID macro. (Read only)
0x4018	FuseChipID6	32	n/a	Value of corresponding fuse bits 95 to 64 of the Custom 112-bit ECID macro. (Read only)
0x401C	FuseChipID7	16	n/a	Value of corresponding fuse bits 111 to 96 of the Custom 112-bit ECID macro. (Read only)

17.3.3   Sub-Block Partition

IBM offer two variants of their ROM macros; A high performance version (ROMHD) and a low power version (ROMLD). It is likely that the low power version will be used unless some implementation issue requires the high performance version. Both versions offer the same bit density. The sub-block partition diagram below does not include the clocking and test signals for the ROM or ECID macros. The CPU subsystem bus interface is described in more detail in section 11.4.3.

17.3.4   Table 100. ROM Block internal signals

Port name	Width	Description
Clocks and Resets		
prst_n	1	Global reset. Synchronous to pclk, active low.
Pclk	1	Global clock

Internal Signals		
rom_adr[11:0]	12	ROM address bus
rom_sel	1	Select signal to the ROM macro instructing it to access the location at rom_adr
rom_oe	1	Output enable signal to the ROM block
rom_data[31:0]	32	Data bus from the ROM macro to the CPU bus interface
rom_dvalid	1	Signal from the ROM macro indicating that the data on rom_data is valid for the address on rom_adr
fuse_data[31:0]	32	Data from the FuseChipID[N] register addressed by fuse_reg_adr
fuse_reg_adr[2:0]	3	Indicates which of the FuseChipID registers is being addressed

Sub-block signal definition

18   Power Safe Storage (PSS) Block

18.1   OVERVIEW

The PSS block provides 128 bytes of storage space that will maintain its state when the rest of the SoPEC device is in sleep mode. The PSS is expected to be used primarily for the storage of decrypted signatures associated with downloaded programmed code but it can also be used to store any information that needs to survive sleep mode (e.g. configuration details). Note that the signature digest only needs to be stored in the PSS before entering sleep mode and the PSS can be used for temporary storage of any data at all other times.

Prior to entering sleep mode the CPU should store all of the information it will need on exiting sleep mode in the PSS. On emerging from sleep mode the boot code in ROM will read the ResetSrc register in the CPR block to determine which reset source caused the wakeup. The reset source information indicates whether or not the PSS contains valid stored data, and the PSS data determines the type of boot sequence to execute. If for any reason a full power-on boot sequence should be performed (e.g. the printer driver has been updated) then this is simply achieved by initiating a full software reset.

Note that a reset or a powerdown (powerdown is implemented by clock gating) of the PSS block will not clear the contents of the 128 bytes of storage. If clearing of the PSS storage is required, then the CPU must write to each location individually.

18.2   IMPLEMENTATION

The storage area of the PSS block will be implemented as a 128-byte register array. The array is located from PSS_base through to PSS_base+0x7F in the address map. The PSS block will only allow read or write accesses with supervisor data space permissions (i.e. cpu_acode[1:0] = 11). All other accesses will result in pss_cpu_berr being asserted. The CPU subsystem bus slave interface is described in more detail in section 11.4.3.

18.2.1   Definitions of I/O

Table 101. PSS Block I/O

216

Port name	Pins	I/O	Description
Clocks and Resets			
prst_n	1	In	Global reset. Synchronous to pclk, active low.
Pclk	1	In	Global clock
CPU Interface			
cpu_adr[6:2]	5	In	CPU address bus. Only 5 bits are required to decode the address space for this block.
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
pss_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpus_rwn	1	In	Common read/not-write signal from the CPU
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
cpu_pss_sel	1	In	Block select from the CPU. When *cpu_pss_sel* is high both *cpu_adr* and *cpu_dataout* are valid
pss_cpu_rdy	1	Out	Ready signal to the CPU. When *pss_cpu_rdy* is high it indicates the last cycle of the access. For a read cycle this means the data on *pss_cpu_data* is valid.
pss_cpu_berr	1	Out	PSS bus error signal to the CPU indicating an invalid access.

19 Low Speed Serial Interface (LSS)

19.1 OVERVIEW

The Low Speed Serial Interface (LSS) provides a mechanism for the internal SoPEC CPU to communicate with external QA chips via two independent LSS buses. The LSS communicates through the GPIO block to the QA chips. This allows the QA chip pins to be reused in multi-SoPEC environments. The LSS Master system-level interface is illustrated in Figure 75. Note that multiple QA chips are allowed on each LSS bus.

19.2 QA COMMUNICATION

The SoPEC data interface to the QA Chips is a low speed, 2 pin, synchronous serial bus. Data is transferred to the QA chips via the *lss_data* pin synchronously with the *lss_clk* pin. When the *lss_clk* is high the data on *lss_data* is deemed to be valid. Only the LSS master in SoPEC can drive the *lss_clk* pin, this pin is an input only to the QA chips. The LSS block must be able to interface with an open-collector pull-up bus. This means that when the LSS block should transmit a logical zero it will drive 0 on the bus, but when it should transmit a logical 1 it will leave high-impedance on the bus (i.e. it doesn't drive the bus). If all the agents on the LSS bus adhere to this protocol then there will be no issues with bus contention.

The LSS block controls all communication to and from the QA chips. The LSS block is the bus master in all cases. The LSS block interprets a command register set by the SoPEC CPU, initiates transactions to the QA chip in question and optionally accepts return data. Any return information is presented through the configuration registers to the SoPEC CPU. The LSS block

5   indicates to the CPU the completion of a command or the occurrence of an error via an interrupt. The LSS protocol can be used to communicate with other LSS slave devices (other than QA chips). However should a LSS slave device hold the clock low (for whatever reason), it will be in violation of the LSS protocol and is not supported. The LSS clock is only ever driven by the LSS master.

10   19.2.1   Start and stop conditions

All transmissions on the LSS bus are initiated by the LSS master issuing a START condition and terminated by the LSS master issuing a STOP condition. START and STOP conditions are always generated by the LSS master. As illustrated in Figure 76, a START condition corresponds to a high to low transition on *lss_data* while *lss_clk* is high. A STOP condition corresponds to a low to

15   high transition on *lss_data* while *lss_clk* is high.

19.2.2   Data transfer

Data is transferred on the LSS bus via a byte orientated protocol. Bytes are transmitted serially. Each byte is sent most significant bit (MSB) first through to least significant bit (LSB) last. One clock pulse is generated for each data bit transferred. Each byte must be followed by an

20   acknowledge bit.

The data on the *lss_data* must be stable during the HIGH period of the *lss_clk* clock. Data may only change when *lss_clk* is low. A transmitter outputs data after the falling edge of *lss_clk* and a receiver inputs the data at the rising edge of *lss_clk*. This data is only considered as a valid data bit at the next *lss_clk* falling edge provided a START or STOP is not detected in the period before

25   the next *lss_clk* falling edge. All clock pulses are generated by the LSS block. The transmitter releases the *lss_data* line (high) during the acknowledge clock pulse (ninth clock pulse). The receiver must pull down the *lss_data* line during the acknowledge clock pulse so that it remains stable low during the HIGH period of this clock pulse.

Data transfers follow the format shown in Figure 77. The first byte sent by the LSS master after a

30   START condition is a primary id byte, where bits 7-2 form a 6-bit primary id (0 is a global id and will address all QA Chips on a particular LSS bus), bit 1 is an even parity bit for the primary id, and bit 0 forms the read/ write sense. Bit 0 is high if the following command is a read to the primary id given or low for a write command to that id. An acknowledge is generated by the QA chip(s) corresponding to the given id (if such a chip exists) by driving the *lss_data* line low

35   synchronous with the LSS master generated ninth *lss_clk*.

19.2.3   Write procedure

The protocol for a write access to a QA Chip over the LSS bus is illustrated in Figure 79 below. The LSS master in SoPEC initiates the transaction by generating a START condition on the LSS bus. It then transmits the primary id byte with a 0 in bit 0 to indicate that the following command is

40   a write to the primary id. An acknowledge is generated by the QA chip corresponding to the given

primary id. The LSS master will clock out M data bytes with the slave QA Chip acknowledging each successful byte written. Once the slave QA chip has acknowledged the Mth data byte the LSS master issues a STOP condition to complete the transfer. The QA chip gathers the M data bytes together and interprets them as a command. See QA Chip Interface Specification for more details on the format of the commands used to communicate with the QA chip[8]. Note that the QA chip is free to not acknowledge any byte transmitted. The LSS master should respond by issuing an interrupt to the CPU to indicate this error. The CPU should then generate a STOP condition on the LSS bus to gracefully complete the transaction on the LSS bus.

19.2.4 Read procedure

The LSS master in SoPEC initiates the transaction by generating a START condition on the LSS bus. It then transmits the primary id byte with a 1 in bit 0 to indicate that the following command is a read to the primary id. An acknowledge is generated by the QA chip corresponding to the given primary id. The LSS master releases the *lss_data* bus and proceeds to clock the expected number of bytes from the QA chip with the LSS master acknowledging each successful byte read. The last expected byte is not acknowledged by the LSS master. It then completes the transaction by generating a STOP condition on the LSS bus. See QA Chip Interface Specification for more details on the format of the commands used to communicate with the QA chip[8].

19.3 IMPLEMENTATION

A block diagram of the LSS master is given in Figure 80. It consists of a block of configuration registers that are programmed by the CPU and two identical LSS master units that generate the signalling protocols on the two LSS buses as well as interrupts to the CPU. The CPU initiates and terminates transactions on the LSS buses by writing an appropriate command to the command register, writes bytes to be transmitted to a buffer and reads bytes received from a buffer, and checks the sources of interrupts by reading status registers.

19.3.1 Definitions of IO

Table 102. LSS IO pins definitions

Port name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
CPU Interface			
cpu_rwn	1	In	Common read/not-write signal from the CPU .
cpu_adr[6:2]	5	In	CPU address bus. Only 5 bits are required to decode the address space for this block
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
cpu_acode[1:0]	2	In	CPU access code signals. cpu_acode[0] - Program (0) / Data (1) access cpu_acode[1] - User (0) / Supervisor (1) access

cpu_lss_sel	1	In	Block select from the CPU. When *cpu_lss_sel* is high both *cpu_adr* and *cpu_dataout* are valid
lss_cpu_rdy	1	Out	Ready signal to the CPU. When *lss_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the LSS block and for a read cycle this means the data on *lss_cpu_data* is valid.
lss_cpu_berr	1	Out	LSS bus error signal to the CPU.
lss_cpu_data[31:0]	32	Out	Read data bus to the CPU
lss_cpu_debug_valid	1	Out	Active high. Indicates the presence of valid debug data on *lss_cpu_data*.
GPIO for LSS buses			
lss_gpio_dout[1:0]	2	Out	LSS bus data output Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
gpio_lss_din[1:0]	2	In	LSS bus data input Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
lss_gpio_e[1:0]	2	Out	LSS bus data output enable, active high Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
lss_gpio_clk[1:0]	2	Out	LSS bus clock output Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
ICU interface			
lss_icu_irq[1:0]	2	Out	LSS interrupt requests Bit 0 - interrupt associated with LSS bus 0 Bit 1 - interrupt associated with LSS bus 1

19.3.2   Configuration registers

The configuration registers in the LSS block are programmed via the CPU interface. Refer to section 11.4 on page 69 for the description of the protocol and timing diagrams for reading and writing registers in the LSS block. Note that since addresses in SoPEC are byte aligned and the CPU only supports 32-bit register reads and writes, the lower 2 bits of the CPU address bus are not required to decode the address space for the LSS block. Table 103 lists the configuration registers in the LSS block. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *lss_cpu_data*.

The input *cpu_acode* signal indicates whether the current CPU access is supervisor, user, program or data. The configuration registers in the LSS block can only be read or written by a supervisor data access, i.e. when *cpu_acode* equals b11. If the current access is a supervisor data access then the LSS responds by asserting *lss_cpu_rdy* for a single clock cycle.

If the current access is anything other than a supervisor data access, then the LSS generates a bus error by asserting *lss_cpu_berr* for a single clock cycle instead of *lss_cpu_rdy* as shown in section 11.4 on page 69. A write access will be ignored, and a read access will return zero.

Table 103. LSS Control Registers

5

Address (LSS_base +)	Register	#bits	Reset	Description
Control registers				
0x00	Reset	1	0x1	A write to this register causes a reset of the LSS.
0x04	LssClockHighLow-Duration	16	0x00C8	*Lss_clk* has a 50:50 duty cycle, this register defines the period of *lss_clk* by means of specifying the duration (in *pclk* cycles) that *lss_clk* is low (or high). The reset value specifies transmission over the LSS bus at a nominal rate of 400kHz, corresponding to a low (or high) duration of 200 *pclk* (160Mhz) cycles. Register should not be set to values less than 8.
0x08	LssClocktoDataHold	6	0x3	Specifies the number of *pclk* cycles that Data must remain valid for after the falling edge of *lss_clk*. Minimum value is 3 cycles, and must to programmed to be less than *LssClockHighLowDuration*.
LSS bus 0 registers				
0x10	Lss0IntStatus	3	0x0	LSS bus 0 interrupt status registers Bit 0 - command completed successfully Bit 1 - error during processing of command, not -acknowledge received after transmission of primary id byte on LSS bus 0 Bit 2 - error during processing of command, not -acknowledge received after transmission of data byte on LSS bus 0 All the bits in *Lss0IntStatus* are cleared when the *Lss0Cmd* register gets written to.

				(Read only register)
0x14	Lss0CurrentState	4	0x0	Gives the current state of the LSS bus 0 state machine. (Read only register). (Encoding will be specified upon state machine implementation)
0x18	Lss0Cmd	21	0x00 _0000	Command register defining sequence of events to perform on LSS bus 0 before interrupting CPU. A write to this register causes all the bits in the *Lss0IntStatus* register to be cleared as well as generating a *lss0_new_cmd* pulse.
0x1C - 0x2C	Lss0Buffer[4:0]	5x32	0x0000 _0000	LSS Data buffer. Should be filled with transmit data before transmit command, or read data bytes received after a valid read command.
LSS bus 1 registers				
0x30	Lss1IntStatus	3 .	0x0	LSS bus 1 interrupt status registers Bit 0 - command completed successfully Bit 1 - error during processing of command, not -acknowledge received after transmission of primary id byte on LSS bus 1 Bit 2 - error during processing of command, not -acknowledge received after transmission of data byte on LSS bus 1 All the bits in *Lss1IntStatus* are cleared when the *Lss1Cmd* register gets written to. (Read only register)
0x34	Lss1CurrentState	4	0x0	Gives the current state of the LSS bus 1 state machine. (Read only register) (Encoding will be specified upon state machine implementation)
0x38	Lss1Cmd	21	0x00_ 0000	Command register defining sequence of events to perform on LSS bus 1 before interrupting CPU. A write to this register causes all the bits in the *Lss1IntStatus* register to be cleared as well as generating a *lss1_new_cmd* pulse.

0x3C - 0x4C	Lss1Buffer[4:0]	5x32	0x0000_0000	LSS Data buffer. Should be filled with transmit data before transmit command, or read data bytes received after a valid read command.
Debug registers				
0x50	LssDebugSel[6:2]	5	0x00	Selects register for debug output. This value is used as the input to the register decode logic instead of *cpu_adr[6:2]* when the LSS block is not being accessed by the CPU, i.e. when *cpu_lss_sel* is 0. The output *lss_cpu_debug_valid* is asserted to indicate that the data on *lss_cpu_data* is valid debug data. This data can be mutliplexed onto chip pins during debug mode.

19.3.2.1 LSS command registers

The LSS command registers define a sequence of events to perform on the respective LSS bus before issuing an interrupt to the CPU. There is a separate command register and interrupt for each LSS bus. The format of the command is given in Table 104. The CPU writes to the command register to initiate a sequence of events on an LSS bus. Once the sequence of events has completed or an error has occurred, an interrupt is sent back to the CPU.

Some example commands are:

- a single START condition (*Start* = 1, *IdByteEnable* = 0, *RdWrEnable* = 0, *Stop* = 0)
- a single STOP condition (*Start* = 0, *IdByteEnable* = 0, *RdWrEnable* = 0, *Stop* = 1)
- a START condition followed by transmission of the id byte (*Start* = 1, *IdByteEnable* = 1, *RdWrEnable* = 0, *Stop* = 0, *IdByte* contains primary id byte)
- a write transfer of 20 bytes from the data buffer (*Start* = 0, *IdByteEnable* = 0, *RdWrEnable* = 1, *RdWrSense* = 0, *Stop* = 0, *TxRxByteCount* = 20)
- a read transfer of 8 bytes into the data buffer (*Start* = 0, *IdByteEnable* = 0, *RdWrEnable* = 1, *RdWrSense* = 1, *ReadNack* = 0, *Stop* = 0, *TxRxByteCount* = 8)
- a complete read transaction of 16 bytes (*Start* = 1, *IdByteEnable* = 1, *RdWrEnable* = 1, *RdWrSense* = 1, *ReadNack* = 1, *Stop* = 1, *IdByte* contains primary id byte, *TxRxByteCount* = 16), etc.

The CPU can thus program the number of bytes to be transmitted or received (up to a maximum of 20) on the LSS bus before it gets interrupted. This allows it to insert arbitrary delays in a transfer at a byte boundary. For example the CPU may want to transmit 30 bytes to a QA chip but insert a delay between the 20th and 21st bytes sent. It does this by first writing 20 bytes to the data buffer. It then writes a command to generate a START condition, send the primary id byte and then transmit the 20 bytes from the data buffer. When interrupted by the LSS block to indicate successful completion of the command the CPU can then write the remaining 10 bytes to the data

buffer. It can then wait for a defined period of time before writing a command to transmit the 10 bytes from the data buffer and generate a STOP condition to terminate the transaction over the LSS bus.

An interrupt to the CPU is generated for one cycle when any bit in *LssNIntStatus* is set. The CPU can read *LssNIntStatus* to discover the source of the interrupt. The *LssNIntStatus* registers are cleared when the CPU writes to the *LssNCmd* register. A null command write to the *LssNCmd* register will cause the *LssNIntStatus* registers to clear and no new command to start. A null command is defined as *Start*, *IdbyteEnable*, *RdWrEnable* and *Stop* all set to zero.

Table 104. LSS command register description

bit(s)	name	Description
0	Start	When 1, issue a START condition on the LSS bus.
1	IdByteEnable	ID byte transmit enable: 1 - transmit byte in *IdByte* field 0 - ignore byte in *IdByte* field
2	RdWrEnable	Read/write transfer enable: 0 - ignore settings of *RdWrSense*, *ReadNack* and *TxRxByteCount* 1 - if RdWrSense is 0, then perform a write transfer of *TxRxByteCount* bytes from the data buffer. if RdWrSense is 1, then perform a read transfer of *TxRxByteCount* bytes into the data buffer. Each byte should be acknowledged and the last byte received is acknowledged/not-acknowledged according to the setting of *ReadNack*.
3	RdWrSense	Read/write sense indicator: 0 - write 1 — read
4	ReadNack	Indicates, for a read transfer, whether to issue an acknowledge or a not-acknowledge after the last byte received (indicated by *TxRxByteCount*). 0 - issue acknowledge after last byte received 1 - issue not-acknowledge after last byte received.
5	Stop	When 1, issue a STOP condition on the LSS bus.
7:6	reserved	Must be 0
15:8	IdByte	Byte to be transmitted if *IdByteEnable* is 1. Bit 8 corresponds to the LSB.

20:16	TxRxByteCount	Number of bytes to be transmitted from the data buffer or the number of bytes to be received into the data buffer. The maximum value that should be programmed is 20, as the size of the data buffer is 20 bytes. Valid values are 1 to 20, 0 is valid when RdWrEnable = 0, other cases are invalid and undefined.

The data buffer is implemented in the LSS master block. When the CPU writes to the *LssNBuffer* registers the data written is presented to the LSS master block via the *lssN_buffer_wrdata* bus and configuration registers block pulses the *lssN_buffer_wen* bit corresponding to the register written. For example if *LssNBuffer[2]* is written to *lssN_buffer_wen[2]* will be pulsed. When the CPU reads the *LssNBuffer* registers the configuration registers block reflect the *lssN_buffer_rdata* bus back to the CPU.

19.3.3   LSS master unit

The LSS master unit is instantiated for both LSS bus 0 and LSS bus 1. It controls transactions on the LSS bus by means of the state machine shown in Figure 83, which interprets the commands that are written by the CPU. It also contains a single 20 byte data buffer used for transmitting and receiving data.

The CPU can write data to be transmitted on the LSS bus by writing to the *LssNBuffer* registers. It can also read data that the LSS master unit receives on the LSS bus by reading the same registers. The LSS master always transmits or receives bytes to or from the data buffer in the same order.

For a transmit command, *LssNBuffer[0][7:0]* gets transmitted first, then *LssNBuffer[0][15:8]*, *LssNBuffer[0][23:16]*, *LssNBuffer[0][31:24]*, *LssNBuffer[1][7:0]* and so on until *TxRxByteCount* number of bytes are transmitted. A receive command fills data to the buffer in the same order. Each new command the buffer start point is reset.

All state machine outputs, flags and counters are cleared on reset. After a reset the state machine goes to the *Reset* state and initialises the LSS pins (*lss_clk* is set to 1, *lss_data* is tristated and allowed to be pulled up to 1). When the reset condition is removed the state machine transitions to the *Wait* state.

It remains in the *Wait* state until *lss_new_cmd* equals 1. If the *Start* bit of the command is 0 the state machine proceeds directly to the *CheckIdByteEnable* state. If the *Start* bit is 1 it proceeds to the *GenerateStart* state and issues a START condition on the LSS bus.

In the *CheckIdByteEnable* state, if the *IdByteEnable* bit of the command is 0 the state machine proceeds directly to the *CheckRdWrEnable* state. If the *IdByteEnable* bit is 1 the state machine enters the *SendIdByte* state and the byte in the *IdByte* field of the command is transmitted on the LSS. The *WaitForIdAck* state is then entered. If the byte is acknowledged, the state machine proceeds to the *CheckRdWrEnable* state. If the byte is not-acknowledged, the state machine proceeds to the *GenerateInterrupt* state and issues an interrupt to indicate a not-acknowledge was received after transmission of the primary id byte.

225

In the *CheckRdWrEnable* state, if the *RdWrEnable* bit of the command is 0 the state machine proceeds directly to the *CheckStop* state. If the *RdWrEnable* bit is 1, *count* is loaded with the value of the *TxRxByteCount* field of the command and the state machine enters either the *ReceiveByte* state if the *RdWrSense* bit of the command is 1 or the *TransmitByte* state if the

5 *RdWrSense* bit is 0.

For a write transaction, the state machine keeps transmitting bytes from the data buffer, decrementing *count* after each byte transmitted, until *count* is 1. If all the bytes are successfully transmitted the state machine proceeds to the *CheckStop* state. If the slave QA chip not-acknowledges a transmitted byte, the state machine indicates this error by issuing an interrupt to

10 the CPU and then entering the *GenerateInterrupt* state.

For a read transaction, the state machine keeps receiving bytes into the data buffer, decrementing *count* after each byte transmitted, until *count* is 1. After each byte received the LSS master must issue an acknowledge. After the last expected byte (i.e. when count is 1) the state machine checks the *ReadNack* bit of the command to see whether it must issue an acknowledge or not-

15 acknowledge for that byte. The *CheckStop* state is then entered.

In the *CheckStop* state, if the *Stop* bit of the command is 0 the state machine proceeds directly to the *GenerateInterrupt* state. If the *Stop* bit is 1 it proceeds to the *GenerateStop* state and issues a STOP condition on the LSS bus before proceeding to the *GenerateInterrupt* state. In both cases an interrupt is issued to indicate successful completion of the command.

20 The state machine then enters the *Wait* state to await the next command. When the state machine reenters the *Wait* state the output pins (*lss_data* and *lss_clk*) are not changed, they retain the state of the last command. This allows the possibility of multi-command transactions. The CPU may abort the current transfer at any time by performing a write to the *Reset* register of the LSS block.

25 *19.3.3.1 START and STOP generation*

START and STOP conditions, which signal the beginning and end of data transmission, occur when the LSS master generates a falling and rising edge respectively on the data while the clock is high.

In the *GenerateStart* state, *lss_gpio_clk* is held high with *lss_gpio_e* remaining deasserted (so the

30 data line is pulled high externally) for *LssClockHighLowDuration* pclk cycles. Then *lss_gpio_e* is asserted and *lss_gpio_dout* is pulled low (to drive a 0 on the data line, creating a falling edge) with *lss_gpio_clk* remaining high for another *LssClockHighLowDuration* pclk cycles.

In the *GenerateStop* state, both *lss_gpio_clk* and *lss_gpio_dout* are pulled low followed by the assertion of *lss_gpio_e* to drive a 0 while the clock is low. After *LssClockHighLowDuration* pclk

35 cycles, *lss_gpio_clk* is set high. After a further *LssClockHighLowDuration* pclk cycles, *lss_gpio_e* is deasserted to release the data bus and create a rising edge on the data bus during the high period of the clock.

If the bus is not in the required state for start and stop generation (*lss_clk*=1, *lss_data*=1 for start, and *lss_clk*=1, *lss_data*=0), the state machine moves the bus to the correct state and proceeds as

described above. Figure 82 shows the transition timing from any bus state to start and stop generation

19.3.3.2 Clock pulse generation

The LSS master holds *lss_gpio_clk* high while the LSS bus is inactive. A clock pulse is generated for each bit transmitted or received over the LSS bus. It is generated by first holding *lss_gpio_clk* low for *LssClockHighLowDuration pclk* cycles, and then high for *LssClockHighLowDuration pclk* cycles.

19.3.3.3 Data De-glitching

When data is received in the LSS block it is passed to a de-glitching circuit. The de-glitch circuit samples the data 3 times on pclk and compares the samples. If all 3 samples are the same then the data is passed, otherwise the data is ignored.

Note that the LSS data input on SoPEC is double registered in the GPIO block before being passed to the LSS.

19.3.3.4 Data reception

The input data, *gpio_lss_di*, is first synchronised to the *pclk* domain by means of two flip-flops clocked by *pclk* (the double register resides in the GPIO block) . The LSS master generates a clock pulse for each bit received. The output *lss_gpio_e* is deasserted *LssClockToDataHold pclk* cycles after the falling edge of *lss_gpio_clk* to release the data bus. The value on the synchronised *gpio_lss_di* is sampled *Tstrobe* number of clock cycles after the rising edge of *lss_gpio_clk* (the data is de-glitched over a further 3 stage register to avoid possible glitch detection). See Figure 84 for further timing information.

In the *ReceiveByte* state, the state machine generates 8 clock pulses. At each *Tstrobe* time after the rising edge of *lss_gpio_clk* the synchronised *gpio_lss_di* is sampled. The first bit sampled is *LssNBuffer[0][7]*, the second *LssNBuffer[0][6]*, etc to *LssNBuffer[0][0]*. For each byte received the state machine either sends an NAK or an ACK depending on the command configuration and the number of bytes received.

In the *SendNack* state the state machine generates a single clock pulse. *lss_gpio_e* is deasserted and the LSS data line is pulled high externally to issue a not-acknowledge.

In the *SendAck* state the state machine generates a single clock pulse. *lss_gpio_e* is asserted and a 0 driven on *lss_gpio_dout* after *lss_gpio_clk* falling edge to issue an acknowledge.

19.3.3.5 Data transmission

The LSS master generates a clock pulse for each bit transmitted. Data is output on the LSS bus on the falling edge of *lss_gpio_clk*.

When the LSS master drives a logical zero on the bus it will assert *lss_gpio_e* and drive a 0 on *lss_gpio_dout* after *lss_gpio_clk* falling edge. *lss_gpio_e* will remain asserted and *lss_gpio_dout* will remain low until the next *lss_clk* falling edge.

When the LSS master drives a logical one *lss_gpio_e* should be deasserted at *lss_gpio_clk* falling edge and remain deasserted at least until the next *lss_gpio_clk* falling edge. This is because the LSS bus will be externally pulled up to logical one via a pull-up resistor.

In the *SendId byte* state, the state machine generates 8 clock pulses to transmit the byte in the *IdByte* field of the current valid command. On each falling edge of *Iss_gpio_clk* a bit is driven on the data bus as outlined above. On the first falling edge *IdByte[7]* is driven on the data bus, on the second falling edge *IdByte[6]* is driven out, etc.

5   In the *TransmitByte* state, the state machine generates 8 clock pulses to transmit the byte at the output of the transmit FIFO. On each falling edge of *Iss_gpio_clk* a bit is driven on the data bus as outlined above. On the first falling edge *LssNBuffer[0][7]* is driven on the data bus, on the second falling edge *LssNBuffer[0][6]* is driven out, etc on to *LssNBuffer[0][7]* bits.

In the *WaitForAck* state, the state machine generates a single clock pulse. At *Tstrobe* time after

10   the rising edge of *Iss_gpio_clk* the synchronized *gpio_Iss_di* is sampled. A 0 indicates an acknowledge and *ack_detect* is pulsed, a 1 indicates a not-acknowledge and *nack_detect* is pulsed.

19.3.3.6 Data rate control

The CPU can control the data rate by setting the clock period of the LSS bus clock by

15   programming appropriate value in *LssClockHighLowDuration*. The default setting for the register is 200 (*pclk* cycles) which corresponds to transmission rate of 400kHz on the LSS bus (the *Iss_clk* is high for *LssClockHighLowDuration* cycles then low for *LssClockHighLowDuration* cycles). The *Iss_clk* will always have a 50:50 duty cycle. The *LssClockHighLowDuration* register should not be set to values less than 8.

20   The hold time of *Iss_data* after the falling edge of *Iss_clk* is programmable by the *LssClocktoDataHold* register. This register should not be programmed to less than 2 or greater than the *LssClockHighLowDuration* value.

19.3.3.7 LSS master timing parameters

The LSS master timing parameters are shown in Figure 84 and the associated values are shown

25   in Table 105 .

Table 105. LSS master timing parameters

Parameter	Description	min	nom	max	unit
LSS Master Driving					
Tp	LSS clock period divided by 2	8	200	FFFF	pclk cycles
Tstart_delay	Time to start data edge from rising clock edge	Tp + *LssClocktoDataHold*			pclk cycles
Tstop_delay	Time to stop data edge from rising clock edge	Tp + *LssClocktoDataHold*			pclk cycles
Tdata_setup	Time from data setup to rising clock edge	Tp - *LssClocktoDataHold*		2	pclk cycles
Tdata_hold	Time from falling clock edge to data hold	*LssClocktoDataHold*			pclk cycles
Tack_setup	Time that outgoing (N)Ack is setup	Tp -		2	pclk cycles

	before *lss_clk* rising edge	*LssClocktoDataHold*			
Tack_hold	Time that outgoing (N)Ack is held after *lss_clk* falling edge	*LssClocktoDataHold*			pclk cycles
LSS Master Sampling					
Tstrobe	LSS master strobe point for incoming data and (N)Ack values	Tp -2		Tp -2	pclk cycles

DRAM SUBSYSTEM

20 DRAM Interface Unit (DIU)

20.1   OVERVIEW

Figure 85 shows how the DIU provides the interface between the on-chip 20 Mbit embedded DRAM and the rest of SoPEC. In addition to outlining the functionality of the DIU, this chapter provides a top-level overview of the memory storage and access patterns of SoPEC and the buffering required in the various SoPEC blocks to support those access requirements.

The main functionality of the DIU is to arbitrate between requests for access to the embedded DRAM and provide read or write accesses to the requesters. The DIU must also implement the initialisation sequence and refresh logic for the embedded DRAM.

The arbitration scheme uses a fully programmable timeslot mechanism for non-CPU requesters to meet the bandwidth and latency requirements for each unit, with unused slots re-allocated to provide best effort accesses. The CPU is allowed high priority access, giving it minimum latency, but allowing bounds to be placed on its bandwidth consumption.

The interface between the DIU and the SoPEC requesters is similar to the interface on PEC1 i.e. separate control, read data and write data busses.

The embedded DRAM is used principally to store:

• CPU program code and data.

• PEP (re)programming commands.

• Compressed pages containing contone, bi-level and raw tag data and header information.

• Decompressed contone and bi-level data.

• Dotline store during a print.

• Print setup information such as tag format structures, dither matrices and dead nozzle information.

20.2   IBM CU-11 EMBEDDED DRAM

20.2.1   Single bank

SoPEC will use the 1.5 V core voltage option in IBM's 0.13 µm class Cu-11 process.

The random read/write cycle time and the refresh cycle time is 3 cycles at 160 MHz [16]. An open page access will complete in 1 cycle if the page mode select signal is clocked at 320 MHz or 2 cycles if the page mode select signal is clocked every 160 MHz cycle. The page mode select signal will be clocked at 160 MHz in SoPEC in order to simplify timing closure. The DRAM word size is 256 bits.

Most SoPEC requesters will make single 256 bit DRAM accesses (see Section 20.4). These accesses will take 3 cycles as they are random accesses i.e. they will most likely be to a different memory row than the previous access.

The entire 20 Mbit DRAM will be implemented as a single memory bank. In Cu-11, the maximum single instance size is 16 Mbit. The first 1 Mbit tile of each instance contains an area overhead so the cheapest solution in terms of area is to have only 2 instances. 16 Mbit and 4Mbit instances would together consume an area of 14.63 mm^2 as would 2 times 10 Mbit instances. 4 times 5 Mbit instances would require 17.2 mm^2.

The instance size will determine the frequency of refresh. Each refresh requires 3 clock cycles. In Cu-11 each row consists of 8 columns of 256-bit words. This means that 10 Mbit requires 5120 rows. A complete DRAM refresh is required every 3.2 ms. Two times 10 Mbit instances would require a refresh every 100 clock cycles, if the instances are refreshed in parallel.

The SoPEC DRAM will be constructed as two 10 Mbit instances implemented as a single memory bank.

20.3   SoPEC MEMORY USAGE REQUIREMENTS

The memory usage requirements for the embedded DRAM are shown in Table 106 .

Table 106. Memory Usage Requirements

Block	Size	Description
Compressed page store	2048 Kbytes	Compressed data page store for Bi-level and contone data
Decompressed Contone Store	108 Kbyte	13824 lines with scale factor 6 = 2304 pixels, store 12 lines, 4 colors = 108 kB 13824 lines with scale factor 5 = 2765 pixels, store 12 lines, 4 colors = 130 kB
Spot line store	5.1 Kbyte	13824 dots/line so 3 lines is 5.1 kB
Tag Format Structure	Typically 12 Kbyte (2.5 mm tags @ 800 dpi)	55 kB in for 384 dot line tags 2.5 mm tags (1/10th inch) @ 1600 dpi require 160 dot lines = 160/384 x55 or 23 kB 2.5 mm tags (1/10th inch) @ 800 dpi require 80/384 x55 = 12 kB
Dither Matrix store	4 Kbytes	64x64 dither matrix is 4 kB 128x128 dither matrix is 16 kB 256x256 dither matrix is 64 kB
DNC Dead Nozzle	1.4 Kbytes	Delta encoded, (10 bit delta position +

Table		6 dead nozzle mask) x% Dnozzle 5% dead nozzles requires (10+6)x 692 Dnoz zles = 1.4 Kbytes
Dot-line store	369.6 Kbytes	Assume each color row is separated by 5 dot lines on the print head The dot line store will be 0+5+10...50+55 = 330 half dot lines + 48 extra half dot lines (4 per dot row) + 60 extra half dot lines estimated to account for printhead misalignment = 438 half dot lines. 438 half dot lines of 6912 dots = 369.6Kbytes
PCU Program code	8 Kbytes	1024 commands of 64 bits = 8 kB
CPU	64 Kbytes	Program code and data
TOTAL	2620 Kbytes (12 Kbyte TFS storage)	

Note:

- Total storage is fixed to 2560 Kbytes to align to 20 Mbit DRAM. This will mean that less space than noted in Table   may be available for the compressed band store.

20.4   SoPEC MEMORY ACCESS PATTERNS

Table 107 shows a summary of the blocks on SoPEC requiring access to the embedded DRAM and their individual memory access patterns. Most blocks will access the DRAM in single 256-bit accesses. All accesses must be padded to 256-bits except for 64-bit CDU write accesses and CPU write accesses. Bits which should not be written are masked using the individual DRAM bit write inputs or byte write inputs, depending on the foundry. Using single 256-bit accesses means that the buffering required in the SoPEC DRAM requesters will be minimized.

Table 107. Memory access patterns of SoPEC DRAM Requesters

DRAM requester	Direction	Memory access pattern
CPU	R	Single 256-bit reads.
	W	Single 32-bit, 16-bit or 8-bit writes.
SCB	R	Single 256-bit reads.
	W	Single 256-bit writes, with byte enables.
CDU	R	Single 256-bit reads of the compressed contone data.
	W	Each CDU access is a write to 4 consecutive DRAM words in the same row but only 64 bits of each word are written with the remaining

		bits write masked. The access time for this 4 word page mode burst is 3 + 2 + 2 +2 = 9 cycles if the page mode select signal is clocked at 160 MHz.
CFU	R	Single 256 bit reads.
LBD	R	Single 256 bit reads.
SFU	R	Separate single 256 bit reads for previous and current line but sharing the same DIU interface
	W	Single 256 bit writes.
TE(TD)	R	Single 256 bit reads. Each read returns 2 times 128 bit tags.
TE(TFS)	R	Single 256 bit reads. TFS is 136 bytes. This means there is unused data in the fifth 256 bit read. A total of 5 reads is required.
HCU	R	Single 256 bit reads. 128 x 128 dither matrix requires 4 reads per line with double buffering. 256 x 256 dither matrix requires 8 reads at the end of the line with single buffering.
DNC	R	Single 256 bit dead nozzle table reads. Each dead nozzle table read con tains 16 dead-nozzle tables entries each of 10 delta bits plus 6 dead nozzle mask bits.
DWU	W	Single 256 bit writes since enable/disable DRAM access per color plane.
LLU	R	Single 256 bit reads since enable/disable DRAM access per color plane.
PCU	R	Single 256 bit reads. Each PCU command is 64 bits so each 256 bit word can contain 4 PCU commands. PCU reads from DRAM used for reprogramming PEP should be executed with minimum latency. If this occurs between pages then there will be free bandwidth as most of the other SoPEC Units will not be requesting from DRAM. If this occurs between bands then the LDB, CDU and TE bandwidth will be free. So the PCU should have a high priority to access to any spare bandwidth.
Refresh		Single refresh.

20.5 BUFFERING REQUIRED IN SoPEC DRAM REQUESTERS

If each DIU access is a single 256-bit access then we need to provide a 256-bit double buffer in the DRAM requester. If the DRAM requester has a 64-bit interface then this can be implemented as an 8 x 64-bit FIFO.

Table 108. Buffer sizes in SoPEC DRAM requesters

DRAM Requester	Direction	Access patterns	Buffering required in block

CPU	R	Single 256-bit reads.	Cache.
	W	Single 32-bit writes but allowing 16-bit or byte addressable writes.	None.
SCB	R	Single 256-bit reads.	Double 256-bit buffer.
	W	Single 256-bit writes, with byte enables.	Double 256-bit buffer.
CDU	R	Single 256-bit reads of the compressed contone data.	Double 256-bit buffer.
	W	Each CDU access is a write to 4 consecutive DRAM words in the same row but only 64 bits of each word are written with the remaining bits write masked.	Double half JPEG block buffer.
CFU	R	Single 256 bit reads.	Triple 256-bit buffer.
LBD	R	Single 256 bit reads.	Double 256-bit buffer.
SFU	R	Separate single 256 bit reads for previous and cur rent line but sharing the same DIU interface	Double 256-bit buffer for each read channel.
	W	Single 256 bit writes.	Double 256-bit buffer.
TE(TD)	R	Single 256 bit reads.	Double 256-bit buffer.
TE(TFS)	R	Single 256 bit reads. TFS is 136 bytes. This means there is unused data in the fifth 256 bit read. A total of 5 reads is required.	Double line-buffer for 136 bytes implemented in TE.
HCU	R	Single 256 bit reads. 128 x 128 dither matrix requires 4 reads per line with double buffering. 256 x 256 dither matrix requires 8 reads at the end of the line with single buffering.	Configurable between dou ble 128 byte buffer and single 256 byte buffer.
DNC	R	Single 256 bit reads	Double 256-bit buffer. Deeper buffering could be specified to cope with local clusters of dead nozzles.
DWU	W	Single 256 bit writes per enabled odd/even color plane.	Double 256-bit buffer per color plane.
LLU	R	Single 256 bit reads per enabled odd/even color plane.	Double 256-bit buffer per color plane.
PCU	R	Single 256 bit reads. Each PCU command is 64 bits so each 256 bit	Single 256-bit buffer.

		DRAM read can contain 4 PCU com mands. Requested command is read from DRAM together with the next 3 contiguous 64-bits which are cached to avoid unnecessary DRAM reads.	
Refresh		Single refresh.	None.

20.6   SoPEC DIU Bandwidth Requirements

Table 109. SoPEC DIU Bandwidth Requirements

Block Name	Direction	Number of cycles between each 256-bit DRAM access to meet peak bandwidth	Peak Bandwidth which must be supplied (bits/cycle)	Average Bandwidth (bits/cycle)	Example number of allocated timeslots[1]
CPU	R				
	W				
SCB	R				
	W	3482	0.734	0.3933	1
CDU	R	128 (SF = 4), 288 (SF = 6), 1:1 compression4	64/n2 (SF=n), 1.8 (SF = 6), 4 (SF = 4) (1:1 compression)	32/10*n2 (SF=n), 0.09 (SF = 6), 0.2 (SF = 4) (10:1 compression)5	1 (SF=6) 2 (SF=4)
	W	For individual accesses: 16 cycles (SF = 4), 36 cycles (SF = 6), n2 cycles (SF=n). Will be implemented as a page mode burst of 4 accesses every 64 cycles (SF = 4), 144 (SF =6), 4*n2 (SF =n) cycles6	64/n2 (SF=n), 1.8 (SF = 6), 4 (SF = 4)	32/n2 (SF=n)7, 0.9 (SF = 6), 2 (SF = 4)	2 (SF=6)8 4 (SF=4)
CFU	R	32 (SF = 4), 48 (SF = 6)9	32/n (SF=n), 5.4 (SF = 6), 8 (SF = 4)	32/n (SF=n), 5.4 (SF = 6), 8 (SF = 4)	6 (SF=6) 8 (SF=4)

LBD	R	256 (1:1 compression)10	1 (1:1 compression)	0.1 (10:1 compression)11	1
SFU	R	12812	2	2	2
	W	25613	1	1	1
TE(TD)	R	25214	1.02	1.02	1
TE(TFS)	R	5 reads per line15	0.093	0.093	0
HCU	R	4 reads per line for 128 x 128 dither matrix16	0.074	0.074	0
DNC	R	106 (5% dead-nozzles 10-bit delta encoded)17	2.4 (clump of dead nozzles)	0.8 (equally spaced dead nozzles)	3
DWU	W	6 writes every 25618	6	6	6
LLU	R	8 reads every 25619	8	6	8
PCU	R	25620	1	1	1
Refresh		10021	2.56	2.56	3 (effective)
TOTAL			SF = 6: 34.9 SF = 4: 41.9 excluding CPU	SF = 6: 27.5 SF = 4: 31.2 excluding CPU	SF = 6: 36 excluding CPU. SF= 4: 41 excluding CPU

Notes:

1: The number of allocated timeslots is based on 64 timeslots each of 1 bit/cycle but broken down to a granularity of 0.25 bit/cycle. Bandwidth is allocated based on peak bandwidth.

2: Wire-speed bandwidth for a 4 wire SCB configuration is 32 Mbits/s for each wire plus 12 Mbit/s for USB. This is a maximum of 138 Mbit/s. The maximum effective data rate is 26 Mbits/s for each wire plus 8 Mbit/s for USB. This is 112 Mbit/s. 112 Mbit/s is 0.734 bits/cycle or 256 bits every 348 cycles.

3: Wire-speed bandwidth for a 2 wire SCB configuration is 32 Mbits/s for each wire plus 12 Mbit/s for USB. This is a maximum of 74 Mbit/s. The maximum effective data rate is 26 Mbits/s for each wire plus 8 Mbit/s for USB. This is 60 Mbit/s. 60 Mbit/s is 0.393 bits/cycle or 256 bits every 650 cycles.

4: At 1:1 compression CDU must read a 4 color pixel (32 bits) every SF^2 cycles.

5: At 10:1 average compression CDU must read a 4 color pixel (32 bits) every $10*SF^2$ cycles.

6: 4 color pixel (32 bits) is required, on average, by the CFU every SF^2 (scale factor) cycles.

The time available to write the data is a function of the size of the buffer in DRAM. 1.5 buffering means 4 color pixel (32 bits) must be written every $SF^2/2$ (scale factor) cycles. Therefore, at a scale factor of SF, 64 bits are required every SF^2 cycles.

Since 64 valid bits are written per 256-bit write (Figure n page379 on page **Error! B   kmark not defined.**) then the DRAM is accessed every SF^2 cycles i.e. at SF4 an access every 16 cycles, at SF6 an access every 36 cycles.

If a page mode burst of 4 accesses is used then each access takes (3 + 2 + 2 +2) equals 9 cycles. This means at SF, a set of 4 back-to-back accesses must occur every $4*SF^2$ cycles. This assumes the page mode select signal is clocked at 160 MHz. CDU timeslots therefore take 9 cycles.

For scale factors lower than 4 double buffering will be used.

7: The peak bandwidth is twice the average bandwidth in the case of 1.5 buffering.

8: Each CDU(W) burst takes 9 cycles instead of 4 cycles for other accesses so CDU timeslots are longer.

9: 4 color pixel (32 bits) read by CFU every SF cycles. At SF4, 32 bits is required every 4 cycles or 256 bits every 32 cycles. At SF6, 32bits every 6 cycles or 256 bits every 48 cycles.

10: At 1:1 compression require 1 bit/cycle or 256 bits every 256 cycles.

11: The average bandwidth required at 10:1 compression is 0.1 bits/cycle.

12: Two separate reads of 1 bit/cycle.

13: Write at 1 bit/cycle.

14: Each tag can be consumed in at most 126 dot cycles and requires 128 bits. This is a maximum rate of 256 bits every 252 cycles.

15: 17 x 64 bit reads per line in PEC1 is 5 x 256 bit reads per line in SoPEC. Double-line buffered storage.

16: 128 bytes read per line is 4 x 256 bit reads per line. Double-line buffered storage.

17: 5% dead nozzles 10-bit delta encoded stored with 6-bit dead nozzle mask requires 0.8 bits/cycle read access or a 256-bit access every 320 cycles. This assumes the dead nozzles are evenly spaced out. In practice dead nozzles are likely to be clumped. Peak bandwidth is estimated as 3 times average bandwidth.

18: 6 bits/cycle requires 6 x 256 bit writes every 256 cycles.

19: 6 bits/160 MHz SoPEC cycle average but will peak at 2 x 6 bits per 106 MHz print head cycle or 8 bits/ SoPEC cycle. The PHI can equalise the DRAM access rate over the line so that the peak rate equals the average rate of 6 bits/ cycle. The print head is clocked at an effective speed of 106 MHz.

20: Assume one 256 read per 256 cycles is sufficient i.e. maximum latency of 256 cycles per access is allowable.

21: Refresh must occur every 3.2 ms. Refresh occurs row at a time over 5120 rows of 2 parallel 10 Mbit instances. Refresh must occur every 100 cycles. Each refresh takes 3 cycles.

20.7   DIU BUS TOPOLOGY

20.7.1   Basic topology

Table 110. SoPEC DIU Requesters

Read	Write	Other
CPU	CPU	Refresh
SCB	SCB	
CDU	CDU	
CFU	SFU	
LBD	DWU	
SFU		
TE(TD)		
TE(TFS)		
HCU		
DNC		
LLU		
PCU		

Table 110 shows the DIU requesters in SoPEC. There are 12 read requesters and 5 write requesters in SoPEC as compared with 8 read requesters and 4 write requesters in PEC1. Refresh is an additional requester.

In PEC1, the interface between the DIU and the DIU requesters had the following main features:

- separate control and address signals per DIU requester multiplexed in the DIU according to the arbitration scheme,

- separate 64-bit write data bus for each DRAM write requester multiplexed in the DIU,

- common 64-bit read bus from the DIU with separate enables to each DIU read requester.

Timing closure for this bussing scheme was straight-forward in PEC1. This suggests that a similar scheme will also achieve timing closure in SoPEC. SoPEC has 5 more DRAM requesters but it will be in a 0.13 um process with more metal layers and SoPEC will run at approximately the same speed as PEC1.

Using 256-bit busses would match the data width of the embedded DRAM but such large busses may result in an increase in size of the DIU and the entire SoPEC chip. The SoPEC requestors would require double 256-bit wide buffers to match the 256-bit busses. These buffers, which must be implemented in flip-flops, are less area efficient than 8-deep 64-bit wide register arrays which can be used with 64-bit busses. SoPEC will therefore use 64-bit data busses. Use of 256-bit busses would however simplify the DIU implementation as local buffering of 256-bit DRAM data would not be required within the DIU.

20.7.1.1 CPU DRAM access

The CPU is the only DIU requestor for which access latency is critical. All DIU write requesters transfer write data to the DIU using separate point-to-point busses. The CPU will use the cpu_dataout[31:0] bus. CPU reads will not be over the shared 64-bit read bus. Instead, CPU reads will use a separate 256-bit read bus.

20.7.2 Making more efficient use of DRAM bandwidth

The embedded DRAM is 256-bits wide. The 4 cycles it takes to transfer the 256-bits over the 64-bit data busses of SoPEC means that effectively each access will be at least 4 cycles long. It takes only 3 cycles to actually do a 256-bit random DRAM access in the case of IBM DRAM.

20.7.2.1 Common read bus

If we have a common read data bus, as in PEC1, then if we are doing back to back read accesses the next DRAM read cannot start until the read data bus is free. So each DRAM read access can occur only every 4 cycles. This is shown in Figure 86 with the actual DRAM access taking 3 cycles leaving 1 unused cycle per access.

20.7.2.2 Interleaving CPU and non-CPU read accesses

The CPU has a separate 256-bit read bus. All other read accesses are 256-bit accesses are over a shared 64-bit read bus. Interleaving CPU and non-CPU read accesses means the effective duration of an interleaved access timeslot is the DRAM access time (3 cycles) rather than 4 cycles.

Figure 87 shows interleaved CPU and non-CPU read accesses.

20.7.2.3 Interleaving read and write accesses

Having separate write data busses means write accesses can be interleaved with each other and with read accesses. So now the effective duration of an interleaved access timeslot is the DRAM access time (3 cycles) rather than 4 cycles. Interleaving is achieved by ordering the DIU arbitration slot allocation appropriately.

Figure 88 shows interleaved read and write accesses. Figure 89 shows interleaved write accesses.

256-bit write data takes 4 cycles to transmit over 64-bit busses so a 256-bit buffer is required in the DIU to gather the write data from the write requester. The exception is CPU write data which is transferred in a single cycle.

Figure 89 shows multiple write accesses being interleaved to obtain 3 cycle DRAM access. Since two write accesses can overlap two sets of 256-bit write buffers and multiplexors to connect two write requestors simultaneously to the DIU are required.

Write requestors only require approximately one third of the total non-CPU bandwidth. This means that a rule can be introduced such that non-CPU write requestors are not allocated adjacent timeslots. This means that a single 256-bit write buffer and multiplexor to connect the one write requestor at a time to the DIU is all that is required.

Note that if the rule prohibiting back-to-back non-CPU writes is not adhered to, then the second write slot of any attempted such pair will be disregarded and re-allocated under the unused read round-robin scheme.

20.7.3   Bus widths summary

Table 111. SoPEC DIU Requesters Data Bus Width

Read	Bus access width	Write	Bus access width

238

CPU	256 (separate)	CPU	32	
SCB	64 (shared)	SCB	64	
CDU	64 (shared)	CDU	64	
CFU	64 (shared)	SFU	64	
LBD	64 (shared)	DWU	64	
SFU	64 (shared)			
TE(TD)	64 (shared)			
TE(TFS)	64 (shared)			
HCU	64 (shared)			
DNC	64 (shared)			
LLU	64 (shared)			
PCU	64 (shared)			

20.7.4   Conclusions

Timeslots should be programmed to maximise interleaving of shared read bus accesses with other accesses for 3 cycle DRAM access. The interleaving is achieved by ordering the DIU arbitration slot allocation appropriately. CPU arbitration has been designed to maximise interleaving with non-CPU requesters

20.8   SoPEC DRAM ADDRESSING SCHEME

The embedded DRAM is composed of 256-bit words. However the CPU-subsystem may need to write individual bytes of DRAM. Therefore it was decided to make the DIU byte addressable. 22 bits are required to byte address 20 Mbit of DRAM.

Most blocks read or write 256 bit words of DRAM. Therefore only the top 17 bits i.e. bits 21 to 5 are required to address 256-bit word aligned locations.

The exceptions are

* CDU which can write 64-bits so only the top 19 address bits i.e. bits 21-3 are required.
* CPU writes can be 8, 16 or 32-bits. The cpu_diu_wmask[1:0] pins indicate whether to write 8, 16 or 32 bits.

All DIU accesses must be within the same 256-bit aligned DRAM word. The exception is the CDU write access which is a write of 64-bits to each of 4 contiguous 256-bit DRAM words.

20.8.1   Write Address Constaints Specific to the CDU

Note the following conditions which apply to the CDU write address, due to the four masked page-mode writes which occur whenever a CDU write slot is arbitrated.

* The CDU address presented to the DIU is cdu_diu_wadr[21:3].
* Bits [4:3] indicate which 64-bit segment out of 256 bits should be written in 4 successive masked page-mode writes.
* Each 10-Mbit DRAM macro has an input address port of width [15:0]. Of these bits, [2:0] are the "page address". Page-mode writes, where you just vary these LSBs (i.e. the "page" or column address), but keep the rest of the address constant, are faster than random writes. This is taken advantage of for CDU writes.

- To guarantee against trying to span a page boundary, the DIU treats "cdu_diu_wadr[6:5]" as being fixed at "00".

- From cdu_diu_wadr[21:3], a initial address of cdu_diu_wadr[21:7] , concatenated with "00", is used as the starting location for the first CDU write. This address is then auto-incremented a further three times.

20.9   DIU PROTOCOLS

The DIU protocols are

- Pipelined i.e. the following transaction is initiated while the previous transfer is in progress.

- Split transaction i.e. the transaction is split into independent address and data transfers.

20.9.1   Read Protocol except CPU

The SoPEC read requestors, except for the CPU, perform single 256-bit read accesses with the read data being transferred from the DIU in 4 consecutive cycles over a shared 64-bit read bus, *diu_data[63:0]*. The read address *<unit>_diu_radr[21:5]* is 256-bit aligned.

The read protocol is:

- *<unit>_diu_rreq* is asserted along with a valid *<unit>_diu_radr[21:5]*.

- The DIU acknowledges the request with *diu_<unit>_rack.* The request should be deasserted. The minimum number of cycles between *<unit>_diu_rreq* being asserted and the DIU generating an *diu_<unit>_rack* strobe is 2 cycles (1 cycle to register the request, 1 cycle to perform the arbitration - see Section 20.14.10).

- The read data is returned on *diu_data[63:0]* and its validity is indicated by *diu_<unit>_rvalid.* The overall 256 bits of data are transferred over four cycles in the order : [63:0] -> [127:64] -> [191:128] -> [255:192].

- When four *diu_<unit>_rvalid* pulses have been received then if there is a further request *<unit>_diu_rreq* should be asserted again. *diu_<unit>_rvalid* will be always be asserted by the DIU for four consecutive cycles. There is a *fixed* gap of 2 cycles between *diu_<unit>_rack* and the first *diu_<unit>_rvalid* pulse. For more detail on the timing of such reads and the implications for back-to-back sequences, see Section 20.14.10.

20.9.2   Read Protocol for CPU

The CPU performs single 256-bit read accesses with the read data being transferred from the DIU over a dedicated 256-bit read bus for DRAM data, *dram_cpu_data[255:0].* The read address *cpu_adr[21:5]* is 256-bit aligned.

The CPU DIU read protocol is:

- *cpu_diu_rreq* is asserted along with a valid *cpu_adr[21:5].*

- The DIU acknowledges the request with *diu_cpu_rack.* The request should be deasserted. The minimum number of cycles between *cpu_diu_rreq* being asserted and the DIU generating a *cpu_diu_rack* strobe is 1 cycle (1 cycle to perform the arbitration - see Section 20.14.10).

- The read data is returned on *dram_cpu_data[255:0]* and its validity is indicated by *diu_cpu_rvalid.*

- When the *diu_cpu_rvalid* pulse has been received then if there is a further request *cpu_diu_rreq* should be asserted again. The *diu_cpu_rvalid* pulse with a gap of 1 cycle after rack (1 cycle for the read data to be returned from the DRAM - see Section 20.14.10).

20.9.3   Write Protocol except CPU and CDU

The SoPEC write requestors, except for the CPU and CDU, perform single 256-bit write accesses with the write data being transferred to the DIU in 4 consecutive cycles over dedicated point-to-point 64-bit write data busses. The write address *<unit>_diu_wadr[21:5]* is 256-bit aligned.

The write protocol is:

- *<unit>_diu_wreq* is asserted along with a valid *<unit>_diu_wadr[21:5]*.
- The DIU acknowledges the request with *diu_<unit>_wack*. The request should be deasserted. The minimum number of cycles between *<unit>_diu_wreq* being asserted and the DIU generating an *diu_<unit>_wack* strobe is 2 cycles (1 cycle to register the request, 1 cycle to perform the arbitration - see Section 20.14.10).
- In the clock cycles following *diu_<unit>_wack* the SoPEC Unit outputs the *<unit>_diu_data[63:0]*, asserting *<unit>_diu_wvalid*. The first *<unit>_diu_wvalid* pulse can occur the clock cycle after *diu_<unit>_wack*. *<unit>_diu_wvalid* remains asserted for the following 3 clock cycles. This allows for reading from an SRAM where new data is available in the clock cycle after the address has changed e.g. the address for the second 64-bits of write data is available the cycle after *diu_<unit>_wack* meaning the second 64-bits of write data is a further cycle later. The overall 256 bits of data is transferred over four cycles in the order : [63:0] -> [127:64] -> [191:128] -> [255:192].
- Note that for SCB writes, each 64-bit quarter-word has an 8-bit byte enable mask associated with it. A different mask is used with each quarter-word. The 4 mask values are transferred along with their associated data, as shown in Figure 92.
- If four consecutive *<unit>_diu_wvalid* pulses are not provided by the requester, then the arbitration logic will disregard the write and re-allocate the slot under the unused read round-robin scheme.

Once all the write data has been output then if there is a further request *<unit>_diu_wreq* should be asserted again.

20.9.4   CPU Write Protocol

The CPU performs single 128-bit writes to the DIU on a dedicated write bus, *cpu_diu_wdata[127:0]*. There is an accompanying write mask, *cpu_diu_wmask[15:0]*, consisting of 16 byte enables and the CPU also supplies a 128-bit aligned write address on *cpu_diu_wadr[21:4]*. Note that writes are *posted* by the CPU to the DIU and stored in a 1-deep buffer. When the DAU subsequently arbitrates in favour of the CPU, the contents of the buffer are written to DRAM.

The CPU write protocol, illustrated in Figure 93., is as follows :-

- The DIU signals to the CPU via *diu_cpu_write_rdy* that its write buffer is empty and that the CPU may post a write whenever it wishes.

- The CPU asserts *cpu_diu_wdatavalid* to enable a write into the buffer and to confirm the validity of the write address, data and mask.
- The DIU de-asserts *diu_cpu_write_rdy* in the following cycle to indicate that its buffer is full and that the posted write is pending execution.
- When the CPU is next awarded a DRAM access by the DAU, the buffer's contents are written to memory. The DIU re-asserts *diu_cpu_write_rdy* once the write data has been captured by DRAM, namely in the "MSN1" DCU state.
- The CPU can then, if it wishes, asynchronously use the new value of *.diu_cpu_write_rdy* to enable a new posted write in the same "MSN1" cycle.

20.9.5   CDU Write Protocol

The CDU performs four 64-bit word writes to 4 contiguous 256-bit DRAM addresses with the first address specified by *cdu_diu_wadr[21:3]*. The write address *cdu_diu_wadr[21:5]* is 256-bit aligned with bits *cdu_diu_wadr[4:3]* allowing the 64-bit word to be selected.

The write protocol is:

- *cdu_diu_wdata* is asserted along with a valid *cdu_diu_wadr[21:3]*.
- The DIU acknowledges the request with *diu_cdu_wack.* The request should be deasserted. The minimum number of cycles between *cdu_diu_wreq* being asserted and the DIU generating an *diu_cdu_wack* strobe is 2 cycles (1 cycle to register the request, 1 cycle to perform the arbitration - see Section 20.14.10).
- In the clock cycles following *diu_cdu_wack* the CDU outputs the *cdu_diu_data[63:0]*, together with asserted *cdu_diu_wvalid*. The first *cdu_diu_wvalid* pulse can occur the clock cycle after *diu_cdu_wack*. *cdu_diu_wvalid* remains asserted for the following 3 clock cycles. This allows for reading from an SRAM where new data is available in the clock cycle after the address has changed e.g. the address for the second 64-bits of write data is available the cycle after *diu_cdu_wack* meaning the second 64-bits of write data is a further cycle later. Data is transferred over the 4-cycle window in an order, such that each successive 64 bits will be written to a monotonically increasing (by 1 location) 256-bit DRAM word.
- If four consecutive *cdu_diu_wvalid* pulses are not provided with the data, then the arbitration logic will disregard the write and re-allocate the slot under the unused read round-robin scheme.
- Once all the write data has been output then if there is a further request *cdu_diu_wreq* . should be asserted again.

20.10   DIU ARBITRATION MECHANISM

The DIU will arbitrate access to the embedded DRAM. The arbitration scheme is outlined in the next sections.

20.10.1   Timeslot based arbitration scheme

Table   summarised the bandwidth requirements of the SoPEC requestors to DRAM. If we allocate the DIU requestors in terms of peak bandwidth then we require 35.25 bits/cycle (at SF =6) and 40.75 bits/ cycle (at SF = 4) for all the requestors except the CPU.

A timeslot scheme is defined with 64 main timeslots. The number of used main timeslots is programmable between 1 and 64.

Since DRAM read requestors, except for the CPU, are connected to the DIU via a 64-bit data bus each 256-bit DRAM access requires 4 $pclk$ cycles to transfer the read data over the shared read bus. The timeslot rotation period for 64 timeslots each of 4 $pclk$ cycles is 256 $pclk$ cycles or 1.6 μs, assuming $pclk$ is 160 MHz. Each timeslot represents a 256-bit access every 256 $pclk$ cycles or 1 bit/cycle. This is the granularity of the majority of DIU requestors bandwidth requirements in Table .

The SoPEC DIU requesters can be represented using 4 bits (Table n page288 on page 268). Using 64 timeslots means that to allocate each timeslot to a requester, a total of 64 x 5-bit configuration registers are required for the 64 main timeslots.

Timeslot based arbitration works by having a pointer point to the current timeslot. When re-arbitration is signaled the arbitration winner is the current timeslot and the pointer advances to the next timeslot. Each timeslot denotes a single access. The duration of the timeslot depends on the access.

Note that advancement through the timeslot rotation is dependent on an enable bit, *RotationSync*, being set. The consequences of clearing and setting this bit are described in section 20.14.12.2.1 on page 295.

If the SoPEC Unit assigned to the current timeslot is not requesting then the unused timeslot arbitration mechanism outlined in Section 20.10.6 is used to select the arbitration winner.

Note that there is *always* an arbitration winner for every slot. This is because the unused read re-allocation scheme includes refresh in its round-robin protocol. If all other blocks are not requesting, an early refresh will act as fall-back for the slot.

20.10.2   Separate read and write arbitration windows

For write accesses, except the CPU, 256-bits of write data are transferred from the SoPEC DIU write requestors over 64-bit write busses in 4 clock cycles. This write data transfer latency means that writes accesses, except for CPU writes and also the CDU, must be arbitrated 4 cycles in advance. (The CDU is an exception because CDU writes can start once the first 64-bits of write data have been transferred since each 64-bits is associated with a write to a different 256-bit word).

Since write arbitration must occur 4 cycles in advance, and the minimum duration of a timeslot duration is 3 cycles, the arbitration rules must be modified to initiate write accesses in advance. Accordingly, there is a write timeslot lookahead pointer shown in Figure 96 two timeslots in advance of the current timeslot pointer.

The following examples illustrate separate read and write timeslot arbitration with no adjacent write timeslots. (Recall rule on adjacent write timeslots introduced in Section 20.7.2.3 on page 238.)

In Figure 97 writes are arbitrated two timeslots in advance. Reads are arbitrated in the same timeslot as they are issued. Writes can be arbitrated in the same timeslot as a read. During arbitration the command address of the arbitrated SoPEC Unit is captured.

Other examples are shown in Figure 98 and Figure 99. The actual timeslot order is always the same as the programmed timeslot order i.e. out of order accesses do not occur and data coherency is never an issue.

Each write must always incur a latency of two timeslots.

5   Startup latency may vary depending on the position of the first write timeslot. This startup latency is not important.

Table 112 shows the 4 scenarios depending on whether the current timeslot and write timeslot lookahead pointers point to read or write accesses.

Table 112. Arbitration with separate windows for read and write accesses

10

current timeslot pointer	write timeslot lookahead pointer	actions
Read	write	Initiate DRAM read, Initiate write arbitration
Read1	read2	Initiate DRAM read1.
Write1	write2	Initiate write2 arbitration. Execute DRAM write1.
Write	read	Execute DRAM write.

If the current timeslot pointer points to a read access then this will be initiated immediately.

If the write timeslot lookahead pointer points to a write access then this access is arbitrated immediately, or immediately after the read access associated·with the current timeslot pointer is

15   initiated.

When a write access is arbitrated the DIU will capture the write address. When the current timeslot pointer advances to the write timeslot then the actual DRAM access will be initiated. Writes will therefore be arbitrated 2 timeslots in advance of the DRAM write occurring.

At initialisation, the write lookahead pointer points to the first timeslot. The current timeslot pointer

20   is invalid until the write lookahead pointer advances to the third timeslot when the current timeslot pointer will point to the first timeslot. Then both pointers advance in tandem.

CPU write accesses are excepted from the lookahead mechanism.

If the selected SoPEC Unit is not requesting then there will be separate read and write selection for unused timeslots. This is described in Section 20.10.6.

25   20.10.3   Arbitration of CPU accesses

What distinguishes the CPU from other SoPEC requestors, is that the CPU requires minimum latency DRAM access i.e. preferably the CPU should get the next available timeslot whenever it requests.

The minimum CPU read access latency is estimated in Table 113. This is the time between the

30   CPU making a request to the DIU and receiving the read data back from the DIU.

244

Table 113. Estimated CPU read access latency ignoring caching

CPU read access latency	Duration
CPU cache miss	1 cycle
CPU MMU logic issues request and DIU arbitration completes	1 cycle
Transfer the read address to the DRAM	1 cycle
DRAM read latency	1 cycle
Register the read data in CPU bridge	1 cycle
Register the read data in CPU	1 cycle
CPU cache miss	1 cycle
CPU MMU logic issues request and DIU arbitration completes	1 cycle
TOTAL gap between requests	6 cycles

If the CPU, as is likely, requests DRAM access again immediately after receiving data from the DIU then the CPU could access every second timeslot if the access latency is 6 cycles. This assumes that interleaving is employed so that timeslots last 3 cycles. If the CPU access latency were 7 cycles, then the CPU would only be able to access every third timeslot.

If a cache hit occurs the CPU does not require DRAM access. For its next DIU access it will have to wait for its next assigned DIU slot. Cache hits therefore will reduce the number of DRAM accesses but not speed up any of those accesses.

To avoid the CPU having to wait for its next timeslot it is desirable to have a mechanism for ensuring that the CPU always gets the next available timeslot without incurring any latency on the non-CPU timeslots.

This can be done by defining each timeslot as consisting of a CPU access preceding a non-CPU access. Each timeslot will last 6 cycles i.e. a CPU access of 3 cycles and a non-CPU access of 3 cycles. This is exactly the interleaving behaviour outlined in Section 20.7.2.2. If the CPU does not require an access, the timeslot will take 3 or 4 and the timeslot rotation will go faster. A summary is given in Table 114.

Table 114. Timeslot access times.

Access	Duration	Explanation
CPU access + non-CPU access	3 + 3 = 6 cycles	Interleaved access
non-CPU access	4 cycles	Access and preceding access both to shared read bus
non-CPU access	3 cycles	Access and preceding access not both to shared read bus
CDU write access	3+2+2+2 = 9 cycles	Page mode select signal is clocked at 160

		MHz

CDU write accesses require 9 cycles. CDU write accesses preceded by a CPU access require 12 cycles. CDU timeslots therefore take longer than all other DIU requestors timeslots.

With a 256 cycle rotation there can be 42 accesses of 6 cycles.

For low scale factor applications, it is desirable to have more timeslots available in the same 256 cycle rotation. So two counters of 4-bits each are defined allowing the CPU to get a maximum of (*CPUPreAccessTimeslots* + 1) pre-accesses for every (*CPUTotalTimeslots* + 1) main slots. A timeslot counter starts at *CPUTotalTimeslots* and decrements every timeslot, while another counter starts at *CPUPreAccessTimeslots* and decrements every timeslot in which the CPU uses its access. When the CPU pre-access counter goes to zero before *CPUTotalTimeslots,* no further CPU accesses are allowed. When the *CPUTotalTimeslots* counter reaches zero both counters are reset to their respective initial values.

The CPU is not included in the list of SoPEC DIU requesters, Table   , for the main timeslot allocations. The CPU cannot therefore be allocated main timeslots. It relies on pre-accesses in advance of such slots as the sole method for DRAM transfers.

CPU access to DRAM can never be fully disabled, since to do so would render SoPEC inoperable. Therefore the *CPUPreAccessTimeslots* and *CPUTotalTimeslots* register values are interpreted as follows : In each succeeding window of (*CPUTotalTimeslots+* 1) slots, the maximum quota of CPU pre-accesses allowed is (*CPUPreAccessTimeslots* + 1). The "+ 1" implementations mean that the CPU quota cannot be made zero.

The various modes of operation are summarised in Table 115 with a nominal rotation period of 256 cycles.

Table 115. CPU timeslot allocation modes with nominal rotation period of 256 cycles

Access Type	Nominal Timeslot duration	Number of timeslots	Notes
CPU Pre-access i.e. *CPUPreAccessTimeslo ts = CPUTotalTimeslots*	6 cycles	42 timeslots	Each access is CPU + non-CPU. If CPU does not use a timeslot then rotation is faster.
Fractional CPU Pre-access i.e. *CPUPreAccessTimeslo ts < CPUTotalTimeslots*	4 or 6 cycles	42-64 timeslots	Each CPU + non-CPU access requires a 6 cycle timeslot.
			Individual non-CPU timeslots take 4 cycles if

			current access and preceding access are both to shared read bus.
			Individual non-CPU timeslots take 3 cycles if current access and preceding access are not both to shared read bus.

20.10.4 CDU accesses

As indicated in Section 20.10.3, CDU write accesses require 9 cycles. CDU write accesses preceded by a CPU access require 12 cycles. CDU timeslots therefore take longer than all other DIU requestors timeslots. This means that when a write timeslot is unused it cannot be re-

5   allocated to a CDU write as CDU accesses take 9 cycles. The write accesses which the CDU write could otherwise replace require only 3 or 4 cycles.

Unused CDU write accesses can be replaced by any other write access according to 20.10.6.1 Unused write timeslots allocation on page 247.

20.10.5 Refresh controller

10   Refresh is not included in the list of SoPEC DIU requesters, Table , for the main timeslot allocations. Timeslots cannot therefore be allocated to refresh.

The DRAM must be refreshed every 3.2 ms. Refresh occurs row at a time over 5120 rows of 2 parallel 10 Mbit instances. A refresh operation must therefore occur every 100 cycles. The *refresh_period* register has a default value of 99. Each refresh takes 3 cycles.

15   A refresh counter will count down the number of cycles between each refresh. When the down-counter reaches 0, the refresh controller will issue a refresh request and the down-counter is reloaded with the value in *refresh_period* and the count-down resumes immediately. Allocation of main slots must take into account that a refresh is required at *least* once every 100 cycles. Refresh is included in the unused read and write timeslot allocation. If unused timeslot allocation

20   results in refresh occurring early by N cycles, then the refresh counter will have counted down to N. In this case, the refresh counter is reset to *refresh_period* and the count-down recommences. Refresh can be preceded by a CPU access in the same way as any other access. This is controlled by the *CPUPreAccessTimeslots* and *CPUTotalTimeslots* configuration registers. Refresh will therefore not affect CPU performance. A sequence of accesses including refresh

25   might therefore be CPU, refresh, CPU, actual timeslot.

20.10.6 Allocating unused timeslots

Unused slots are re-allocated separately depending on whether the unused access was a read access or a write access. This is best-effort traffic. Only unused non-CPU accesses are re-allocated.

30   *20.10.6.1 Unused write timeslots allocation*

Unused write timeslots are re-allocated according to a *fixed* priority order shown in Table 116 .

Table 116. Unused write timeslot priority order

Name	Priority Order
SCB(W)	1
SFU(W)	2
DWU	3
Unused read timeslot allocation	4

CDU write accesses cannot be included in the unused timeslot allocation for write as CDU accesses take 9 cycles. The write accesses which the CDU write could otherwise replace require only 3 or 4 cycles.

5   Unused write timeslot allocation occurs two timeslots in advance as noted in Section 20.10.2. If the units at priorities 1-3 are not requesting then the timeslot is re-allocated according to the unused read timeslot allocation scheme described in Section 20.10.6.2. However, the unused read timeslot allocation will occur when the current timeslot pointer of Figure 96 reaches the timeslot i.e. it will not occur in advance.

10   *20.10.6.2 Unused read timeslots allocation*

Unused read timeslots are re-allocated according to a two level round-robin scheme. The SoPEC Units included in read timeslot re-allocation is shown in Table 117.

15   Table 117. Unused read timeslot allocation

Name
SCB(R)
CDU(R)
CFU
LBD
SFU(R)
TE(TD)
TE(TFS)
HCU
DNC
LLU
PCU
CPU / Refresh

Each SoPEC requestor has an associated bit, *ReadRoundRobinLevel*, which indicates whether it is in level 1 or level 2 round-robin.

Table 118. Read round-robin level selection

Level	Action
ReadRoundRobinLevel = 0	Level 1
ReadRoundRobinLevel = 1	Level 2

A pointer points to the most recent winner on each of the round-robin levels. Re-allocation is carried out by traversing level 1 requesters, starting with the one immediately succeeding the last level 1 winner. If a requesting unit is found, then it wins arbitration and the level 1 pointer is shifted to its position. If no level 1 unit wants the slot, then level 2 is similarly examined and its pointer adjusted.

Since refresh occupies a (shared) position on one of the two levels and continually requests access, there will always be some round-robin winner for any unused slot.

20.10.6.2.1   Shared CPU / Refresh Round-Robin Position

Note that the CPU can conditionally be allowed to take part in the unused read round-robin scheme. Its participation is controlled via the configuration bit *EnableCPURoundRobin*. When this bit is set, the CPU and refresh *share* a joint position in the round-robin order, shown in Table . When cleared, the position is occupied by refresh alone.

If the shared position is next in line to be awarded an unused non-CPU read/write slot, then the CPU will have first option on the slot. Only if the CPU doesn't want the access, will it be granted to refresh. If the CPU is excluded from the round robin, then any awards to the position benefit refresh.

20.11   GUIDELINES FOR PROGRAMMING THE DIU

Some guidelines for programming the DIU arbitration scheme are given in this section together with an example.

20.11.1   Circuit Latency

Circuit latency is a fixed service delay which is incurred, as and from the acceptance by the DIU arbitration logic of a block's pending read/write request. It is due to the processing time of the request, readying the data, plus the DRAM access time. Latencies differ for read and write requests. See Tables 79 and 80 for respective breakdowns.

If a requesting block is currently stalled, then the *longest* time it will have to wait between issuing a new request for data and actually receiving it would be its timeslot period, plus the circuit latency overhead, along with any intervening non-standard slot durations, such as refresh and CDU(W). In any case, a stalled block will always incur this latency as an additional overhead, when coming out of a stall.

In the case where a block starts up or unstalls, it will start processing newly-received data at a time beyond its serviced timeslot equivalent to the circuit latency. If the block's timeslots are evenly spaced apart in time to match its processing rate, (in the hope of minimising stalls,) then

the earliest that the block could restall, if not re-serviced by the DIU, would be the same latency delay beyond its next timeslot occurrence. Put another way, the latency incurred at start-up pushes the potential DIU-induced stall point out by the same fixed delta beyond each successive timeslot allocated to the block. This assumes that a block re-requests access well in advance of its upcoming timeslots. Thus, for a given stall-free run of operation, the circuit latency overhead is only incurred inititially when unstalling.

While a block can be stalled as a result of how quickly the DIU services its DRAM requests, it is also prone to stalls caused by its upstream or downstream neighbours being able to supply or consume data which is transferred between the blocks directly, (as opposed to via the DIU). Such neighbour-induced stalls, often occurring at events like end of line, will have the effect that a block's DIU read buffer will tend to fill, as the block stops processing read data. Its DIU write buffer will also tend to fill, unable to despatch to DRAM until the downstream block frees up shared-access DRAM locations. This scenario is beneficial, in that when a block unstalls as a result of its neighbour releasing it, then that block's read/write DIU buffers will have a fill state less likely to stall it a second time, as a result of DIU service delays.

A block's slots should be scheduled with a *service guarantee* in mind. This is dictated by the block's processing rate and hence, required access to the DRAM. The rate is expressed in terms of bits per cycle across a processing window, which is typically (though not always) 256 cycles. Slots should be evenly interspersed in this window (or "rotation") so that the DIU can fulfill the block's service needs.

The following ground rules apply in calculating the distribution of slots for a given non-CPU block:-

- The block can, at maximum, suffer a stall *once* in the rotation, (i.e. unstall and restall) and hence incur the circuit latency described above.

This rule is, by definition, always fulfilled by those blocks which have a service requirement of only 1 bit/cycle (equivalent to 1 slot/rotation) or fewer. It can be shown that the rule is also satisfied by those blocks requiring more than 1 bit/cycle. See Section 20.12.1 Slot Distributions and Stall Calculations for Individual Blocks, on page 255.

- Within the rotation, certain slots will be unavailable, due to their being used for refresh. (See Section 20.11.2 **Refresh latencies**)

- In programming the rotation, account must be taken of the fact that any CDU(W) accesses will consume an extra 6 cycles/access, over and above the norm, in CPU pre-access mode, or 5 cycles/access without pre-access.

- The total delay overhead due to latency, refreshes and CDU(W) can be factored into the service guarantee for *all* blocks in the rotation by deleting *once*, (i.e. reducing the rotation window,) that number of slots which equates to the cumulative duration of these various anomalies.

- The use of lower scale factors will imply a more frequent demand for slots by non-CPU blocks. The percentage of slots in the overall rotation which can therefore be designated as CPU pre-access ones should be calculated last, based on what can be accommodated in the light of the non-CPU slot need.

Read latency is summarised below in Table 119 .

Table 119. Read latency

Non-CPU read access latency	Duration
non-CPU read requestor internally generates DIU request	1 cycle
register the non- CPU read request	1 cycle
complete the arbitration of the request	1 cycle
transfer the read address to the DRAM	1 cycle
DRAM read latency	1 cycle
register the DRAM read data in DIU	1 cycle
register the 1st 64-bits of read data in requester	1 cycle
register the 2nd 64-bits of read data in requester	1 cycle
register the 3rd 64-bits of read data in requester	1 cycle
register the 4th 64-bits of read data in requester	1 cycle
TOTAL	10 cycles

5   Write latency is summarised in Table 120.

Table 120. Write latency

Non-CPU write access latency	Duration
non-CPU write requestor internally generates DIU request	1 cycle
register the non-CPU write request	1 cycle
complete the arbitration of the request	1 cycle
transfer the acknowledge to the write requester	1 cycle
transfer the 1st 64 bits of write data to the DIU	1 cycle
transfer the 2nd 64 bits of write data to the DIU	1 cycle
transfer the 3rd 64 bits of write data to the DIU	1 cycle
transfer the 4th 64 bits of write data to the DIU	1 cycle
Write to DRAM with locally registered write data	1 cycle
TOTAL	9 cycles

Timeslots removed to allow for read latency will also cover write latency, since the former is the

10   larger of the two.

251

20.11.2 Refresh latencies

The number of allocated timeslots for each requester needs to take into account that a refresh must occur every 100 cycles. This can be achieved by deleting timeslots from the rotation since the number of timeslots is made programmable.

Refresh is preceded by a CPU access in the same way as any other access. This is controlled by the *CPUPreAccessTimeslots* and *CPUTotalTimeslots* configuration registers. Refresh will therefore not affect CPU performance.

As an example, in CPU pre-access mode each timeslot will last 6 cycles. If the timeslot rotation has 50 timeslots then the rotation will last 300 cycles. The refresh controller will trigger a refresh every 100 cycles. Up to 47 timeslots can be allocated to the rotation ignoring refresh. Three timeslots deleted from the 50 timeslot rotation will allow for the latency of a refresh every 100 cycles.

20.11.3 Ensuring sufficient DNC and PCU access

PCU command reads from DRAM are exceptional events and should complete in as short a time as possible. Similarly, we must ensure there is sufficient free bandwidth for DNC accesses e.g. when clusters of dead nozzles occur. In Table   DNC is allocated 3 times average bandwidth. PCU and DNC can also be allocated to the level 1 round-robin allocation for unused timeslots so that unused timeslot bandwidth is preferentially available to them.

20.11.4 Basing timeslot allocation on peak bandwidths

Since the embedded DRAM provides sufficient bandwidth to use 1:1 compression rates for the CDU and LBD, it is possible to simplify the main timeslot allocation by basing the allocation on peak bandwidths. As combined bi-level and tag bandwidth at 1:1 scaling is only 5 bits/cycle, we will usually only consider the contone scale factor as the variable in determining timeslot allocations.

If slot allocation is based on peak bandwidth requirements then DRAM access will be *guaranteed* to all SoPEC requesters. If we do not allocate slots for peak bandwidth requirements then we can also allow for the peaks *deterministically* by adding some cycles to the print line time.

20.11.5 Adjacent timeslot restrictions

20.11.5.1 Non-CPU write adjacent timeslot restrictions

Non-CPU write requestors should not be assigned adjacent timeslots as described in Section 20.7.2.3. This is because adjacent timeslots assigned to non-CPU requestors would require two sets of 256-bit write buffers and multiplexors to connect two write requestors simultaneously to the DIU. Only one 256-bit write buffer and multiplexor is implemented. Recall from section 20.7.2.3 on page 238 that if adjacent non-CPU writes are attempted, that the second write of any such pair will be disregarded and re-allocated under the unused read scheme. .

20.11.5.2 Same DIU requestor adjacent timeslot restrictions

All DIU requesters have state-machines which request and transfer the read or write data before requesting again. From Figure 90 read requests have a minimum separation of 9 cycles. From Figure 92 write requests have a minimum separation of 7 cycles. Therefore adjacent timeslots

should not be assigned to a particular DIU requester because the requester will not be able to make use of all these slots.

In the case that a CPU access precedes a non-CPU access timeslots last 6 cycles so write and read requesters can only make use of every second timeslot. In the case that timeslots are not preceded by CPU accesses timeslots last 4 cycles so the same write requester can use every second timeslot but the same read requestor can use only every third timeslot. Some DIU requestors may introduce additional pipeline delays before they can request again. Therefore timeslots should be separated by more than the minimum to allow a margin.

20.11.6   Line margin

The SFU must output 1 bit/cycle to the HCU. Since *HCUNumDots* may not be a multiple of 256 bits the last 256-bit DRAM word on the line can contain extra zeros. In this case, the SFU may not be able to provide 1 bit/cycle to the HCU. This could lead to a stall by the SFU. This stall could then propagate if the margins being used by the HCU are not sufficient to hide it. The maximum stall can be estimated by the calculation: DRAM service period - X scale factor * dots used from last DRAM read for HCU line.

Similarly, if the line length is not a multiple of 256-bits then e.g. the LLU could read data from DRAM which contains padded zeros. This could lead to a stall. This stall could then propagate if the page margins cannot hide it.

A single addition of 256 cycles to the line time will suffice for all DIU requesters to mask these stalls.

20.12   EXAMPLE OUTLINE DIU PROGRAMMING

Table 121. Timeslot allocation based on peak bandwidth

Block Name	Direction	Peak Bandwidth which must be supplied (bits/cycle)	MainTimeslots allocated
SCB	R		
	W	0.734[7]	1
CDU	R	0.9 (SF = 6), 2 (SF = 4)	1 (SF = 6) 2 (SF = 4)
	W	1.8 (SF = 6), [8] 4 (SF = 4)	2 (SF = 6) 4 (SF = 4)
CFU	R	5.4 (SF = 6), 8 (SF = 4)	6 (SF = 6) 8 (SF = 4)

---

[7]   The SCB figure of 0.734 bits/cycle applies to *multi*-SoPEC systems. For *single*-SoPEC systems, the figure is 0.050 bits/cycle.

[8]   Bandwidth for CDU(W) is *peak* value. Because of 1.5 buffering in DRAM, peak CDU(W) b/w equals 2 x average CDU(W) b/w. For CDU(R), peak b/w = average CDU(R) b/w.

LBD	R	1	1
SFU	R	2	2
	W	1	1
TE(TD)	R	1.02	1
TE(TFS)	R	0.093	0
HCU	R	0.074	0
DNC	R	2.4	3
DWU	W	6	6
LLU .	R	8	8
PCU	R	1	1
TOTAL			33 (SF=6) 38 (SF=4)

Table 121 shows an allocation of main timeslots based on the peak bandwidths of Table .
The bandwidth required for each unit is calculated allowing extra cycles for read and write circuit
latency for each access requiring a bandwidth of more than 1 bit/cycle. Fractional bandwidth is
supplied via unused read slots.

5   The timeslot rotation is 256 cycles. Timeslots are deleted from the rotation to allow for circuit
latencies for accesses of up to 1 bit per cycle i.e. 1 timeslot per rotation.

Example 1: Scale-factor = 6

Program the *MainTimeslot* configuration register (Table) for peak required bandwidths of
SoPEC Units according to the scale factor.

10   Program the read round-robin allocation to share unused read slots. Allocate PCU, DNC, HCU
and TFS to level 1 read round-robin.

- Assume scale-factor of 6 and peak bandwidths from Table .

  - Assign all DIU requestors except TE(TFS) and HCU to multiples of 1 timeslot, as indicated
    in Table , where each timeslot is 1 bit/cycle. This requires 33 timeslots.

15 - No timeslots are explicitly allocated for the fractional bandwidth requirements of TE(TFS)
    and HCU accesses. Instead, these units are serviced via unused read slots.

  - Allow 3 timeslots to allow for 3 refreshes in the rotation.

  - Therefore, 36 scheduled slots are used in the rotation for main timeslots and refreshes,
    some or all of which may be able to have a CPU pre-access, provided they fit in the rotation

20   window.

  - Each of the 2 CDU(W) accesses requires 9 cycles. Per access, this implies an overhead of
    1 slot (12 cycles instead of 6) in pre-access mode, or 1.25 slots (9 cycles instead of 4) for
    no pre-access. The cumulative overhead of the two accesses is either 2 slots (pre-access)
    or 3 slots (no pre-access).

25 - Assuming all blocks require a service guarantee of no more than a single stall across 256
    bits, allow 10 cycles for read latency, which also takes care of 9-cycle write latency. This
    can be accounted for by reserving 2 six-cycle slots (CPU pre-access) or 3 four-cycle slots
    (no pre-access).

- Assume a 256 cycle timeslot rotation.
- CDU(W) and read latency reduce the number of available cycles in a rotation to: 256 - 2x6 - 2x6 = 232 cycles (CPU pre-access) or 256 - 3x4 - 3x4 = 232 cycles (no pre-access).
- As a result, 232 cycles available for 36 accesses implies each access can take 232 / 36 = 6.44 cycles maximum. So, all accesses can have a pre-access.
- Therefore the CPU achieves a pre-access ratio of 36 / 36 = 100% of slots in the rotation.

Example 2: Scale-factor = 4

Program the *MainTimeslot* configuration register (Table   ) for peak required bandwidths of SoPEC Units according to the scale factor. Program the read round-robin allocation to share unused read slots. Allocate PCU, DNC, HCU and TFS to level 1 read round-robin.

- Assume scale-factor of 4 and peak bandwidths from Table   .
- Assign all DIU requestors except TE(TFS) and HCU multiples of 1 timeslot, as indicated in Table   , where each timeslot is 1 bit/cycle. This requires 38 timeslots.
- No timeslots are explicitly allocated for the fractional bandwidth requirements of TE(TFS) and HCU accesses. Instead, these units are serviced via unused read slots.
- Allow 3 timeslots to allow for 3 refreshes in the rotation.
- Therefore, 41 scheduled slots are used in the rotation for main timeslots and refreshes, some or all of which can have a CPU pre-access, provided they fit in the rotation window.
- Each of the 4 CDU(W) accesses requires 9 cycles. Per access, this implies an overhead of 1 slot (12 cycles instead of 6) for pre-access mode, or 1.25 slots (9 cycles instead of 4) for no pre-access. The cumulative overhead of the four accesses is either 4 slots (pre-access) or 5 slots (no pre-access).
- Assuming all blocks require a service guarantee of no more than a single stall across 256 bits, allow 10 cycles for read latency, which also takes care of 9-cycle write latency. This can be accounted for by reserving 2 six-cycle slots (CPU pre-access) or 3 four-cycle slots (no pre-access).
- Assume a 256 cycle timeslot rotation.
- CDU(W) and read latency reduce the number of available cycles in a rotation to: 256 - 4x6 - 2x6 = 220 cycles (CPU pre-access) or 256 - 5x4 - 3x4 = 224 cycles (no pre-access).
- As a result, between 220 and 224 cycles are available for 41 accesses, which implies each access can take between 220 / 41 = 5.36 cycles and 224 / 41 = 5.46 cycles.
- Work out how many slots can have a pre-access: For the lower number of 220 cycles, this implies   (41 - n)*6 + n*4 <= 220, where n = number of slots with no pre-access cycle. Solving the equation gives n >= 13. Check answer: 28*6 + 13*4 = 220.
- So 28 slots out of the 41 in the rotation can have CPU pre-accesses.
- The CPU thus achieves a pre-access ratio of 28 / 41 = 68.3% of slots in the rotation.

20.12.1   Slot Distributions and Stall Calculations for Individual Blocks

The following sections show how the slots for blocks with a service requirement greater than 1 bit/cycle should be distributed. Calculations are included to check that such blocks will not suffer more than one stall per rotation.

20.12.1.1 SFU

This has 2 bits/cycle on read but this is two separate channels of 1 bit/cycle sharing the same DIU interface so it is effectively 2 channels each of 1 bit/cycle so allowing the same margins as the LBD will work.

20.12.1.2 DWU

The DWU has 12 double buffers in each of the 6 colour planes, odd and even. These buffers are filled by the DNC and will request DIU access when double buffers fill. The DNC supplies 6 bits to the DWU every cycle (6 odd in one cycle, 6 even in the next cycle). So the service deadline is 512 cycles, given 6 accesses per 256-cycle rotation.

20.12.1.3 CFU

Here the requirement is that the DIU stall should be less than the time taken for the CFU to consume one third of its triple buffer. The total DIU stall = refresh latency + extra CDU(W) latency + read circuit latency = 3 + 5 (for 4 cycle timeslots) + 10 = 18 cycles. The CFU can consume its data at 8 bits/cycle at SF = 4. Therefore 256 bits of data will last 32 cycles so the triple buffer is safe. In fact we only need an extra 144 bits of buffering or 3 x 64 bits. But it is safer to have the full extra 256 bits or 4 x 64 bits of buffering.

20.12.1.4 LLU

The LLU has 2 channels, each of which could request at 6 bits/106 MHz channel or 4 bits/160MHz cycle, giving a total of 8 bits/160MHz cycle. The service deadline for each channel is 256 x 106 MHz cycles, i.e. all 6 colours must be transferred in 256 cycles to feed the printhead. This equates to 384 x 160 MHz cycles.

Over a span of 384 cycles, there will be 6 CDU(W) accesses, 4 refreshes and one read latency encountered at most. Assuming CPU pre-accesses for these occurrences, this means the number of available cycles is given by 384 - 6x6 - 4x6 - 10 = 314 cycles.

For a CPU pre-access slot rate of 50%, 314 cycles implies 31 CPU and 63 non-CPU accesses (31 x 6 + 32 x 4 = 314). For 12 LLU accesses interspersed amongst these 63 non-CPU slots, implies an LLU allocation rate of approximately one slot in 5.

If the CPU pre-access is 100% across all slots, then 314 cycles gives 52 slots each to CPU and non-CPU accesses, (52 x 6 = 312 cycles). Twelve accesses spread over 52 slots, implies a 1-in-4 slot allocation to the LLU.

The same LLU slot allocation rate (1 slot in 5, or 1 in 4) can be applied to programming slots across a *256-cycle* rotation window. The window *size* does not affect the occurrence of LLU slots, so the 384-cycle service requirement will be fulfilled.

20.12.1.5 DNC

This has a 2.4 bits/cycle bandwidth requirement. Each access will see the DIU stall of 18 cycles. 2.4 bits/ cycle corresponds to an access every 106 cycles within a 256 cycle rotation. So to allow for DIU latency we need an access every 106 -18 or 88 cycles. This is a bandwidth of 2.9 bits/cycle, requiring 3 timeslots in the rotation.

20.12.1.6 CDU

The JPEG decoder produces 8 bits/cycle. Peak CDUR[ead] bandwidth is 4 bits/cycle (SF=4), peak CDUW[rite] bandwidth is 4 bits/cycle (SF=4). both with 1.5 DRAM buffering.

The CDU(R) does a DIU read every 64 cycles at scale factor 4 with 1.5 DRAM buffering. The delay in being serviced by the DIU could be read circuit latency (10) + refresh (3) + extra CDU(W) cycles (6) = 19 cycles. The JPEG decoder can consume each 256 bits of DIU-supplied data at 8 bits/cycle, i.e. in 32 cycles. If the DIU is 19 cycles late (due to latency) in supplying the read data then the JPEG decoder will have finished processing the read data 32 + 19 = 49 cycles after the DIU access. This is 64 - 49 = 15 cycles in advance of the next read. This 15 cycles is the upper limit on how much the DIU read service can further be delayed, without causing a stall. Given this margin, a stall on the read side will not occur.

On the write side, for scale factor 4, the access pattern is a DIU writes every 64 cycles with 1.5 DRAM buffereing. The JPEG decoder runs at 8 bits cycle and consumes 256 bits in 32 cycles. The CDU will not stall if the JPEG decode time (32) + DIU stall (19) < 64, which is true.

20.13 CPU DRAM ACCESS PERFORMANCE

The CPU's share of the timeslots can be specified in terms of guaranteed bandwidth and average bandwidth allocations.

The CPU's access rate to memory depends on

- the CPU read access latency i.e. the time between the CPU making a request to the DIU and receiving the read data back from the DIU.
- how often it can get access to DIU timeslots.

Table estimated the CPU read latency as 6 cycles.

How often the CPU can get access to DIU timeslots depends on the access type. This is summarised in Table 122 .

Table 122. CPU DRAM access performance

Access Type	Nominal Timeslot Duration	CPU DRAM access rate	Notes
CPU Pre-access	6 cycles	Lower bound (guaranteed bandwidth) is 160 MHz / 6 = 26.27 MHz	CPU can access every timeslot.
Fractional CPU Pre-access	4 or 6 cycles	Lower bound (guaranteed bandwidth) is $(160\ MHz * N / P)$	CPU accesses precede a fraction N of timeslots where $N = C/T$. $C = CPUPreAccessTimeslots$ $T = CPUTotalTimeslots$ $P = (6^*C + 4^*(T-C)) / T$

In both CPU Pre-access and Fractional CPU Pre-access modes, if the CPU is not requesting the timeslots will have a duration of 3 or 4 cycles depending on whether the current access and preceding access are both to the shared read bus. This will mean that the timeslot rotation will run faster and more bandwidth is available.

5 If the CPU runs out of its instruction cache then instruction fetch performance is only limited by the on-chip bus protocol. If data resides in the data cache then 160 MHz performance is achieved. Accessing memory mapped registers, PSS or ROM with a 3 cycle bus protocol (address cycle + data cycle) gives 53 MHz performance.

Due to the action of CPU caching, some bandwidth limiting of the CPU in Fractional CPU Pre-

10 access mode is expected to have little or no impact on the overall CPU performance.

20.14   IMPLEMENTATION

The DRAM Interface Unit (DIU) is partitioned into 2 logical blocks to facilitate design and verification.

a. The DRAM Arbitration Unit (DAU) which interfaces with the SoPEC DIU requesters.

15 b. The DRAM Controller Unit (DCU) which accesses the embedded DRAM.

The basic principle in design of the DIU is to ensure that the eDRAM is accessed at its maximum rate while keeping the CPU read access latency as low as possible.

The DCU is designed to interface with single bank 20 Mbit IBM Cu-11 embedded DRAM performing random accesses every 3 cycles. Page mode burst of 4 write accesses, associated

20 with the CDU, are also supported.

The DAU is designed to support interleaved accesses allowing the DRAM to be accessed every 3 cycles where back-to-back accesses do not occur over the shared 64-bit read data bus.

20.14.1   DIU Partition

20.14.2   Definition of DCU IO

25   Table 123. DCU interface

Port Name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	SoPEC Functional clock
dau_dcu_reset_n	1	In	Active-low, synchronous reset in *pclk* domain. Incorporates DAU hard and soft resets.
Inputs from DAU			
dau_dcu_msn2stall	1	In	Signal indicating from DAU Arbitration Logic which when asserted stalls DCU in *MSN2* state.
dau_dcu_adr[21:5]	17	In	Signal indicating the address for the DRAM access. This is a 256-bit aligned DRAM address.
dau_dcu_rwn	1	In	Signal indicating the direction for the DRAM

			access (1=read, 0=write).
dau_dcu_cduwpage	1	In	Signal indicating if access is a CDU write page mode access (1=CDU page mode, 0=not CDU page mode).
dau_dcu_refresh	1	In	Signal indicating that a refresh command is to be issued. If asserted *dau_dcu_adr*, *dau_dcu_rwn* and *dau_dcu_cduwpage* are ignored.
dau_dcu_wdata	256	In	256-bit write data to DCU
dau_dcu_wmask	32	In	Byte encoded write data mask for 256-bit *dau_dcu_wdata* to DCU Polarity : A "1" in a bit field of *dau_dcu_wmask* means that the corresponding byte in the 256-bit *dau_dcu_wdata* <u>is</u> written to DRAM.
Outputs to DAU			
dcu_dau_adv	1	Out	Signal indicating to DAU to supply next command to DCU
dcu_dau_wadv	1	Out	Signal indicating to DAU to initiate next non-CPU write
dcu_dau_refreshcomplete	1	Out	Signal indicating that the DCU has completed a refresh.
dcu_dau_rdata	256	Out	256-bit read data from DCU.
dcu_dau_rvalid	1	Out	Signal indicating valid read data on *dcu_dau_rdata*.

20.14.3 DRAM access types

The DRAM access types used in SoPEC are summarised in Table 124. For a refresh operation the DRAM generates the address internally.

Table 124. SoPEC DRAM access types

5

Type	Access
Read	Random 256-bit read
Write	Random 256-bit write with byte write masking.
	Page mode write for burst of 4 256-bit words with byte write masking
Refresh	Single refresh

20.14.4 Constructing the 20 Mbit DRAM from two 10 Mbit instances

The 20 Mbit DRAM is constructed from two 10 Mbit instances. The address ranges of the two instances are shown in Table 125 :

Table 125. Address ranges of the two 10 Mbit instances in the 20 Mbit DRAM

Instance	Address	Hex 256-bit word address	Binary 256-bit word address
Instance0	First word in lower 10 Mbit	00000	0 0000 0000 0000 0000
Instance0	Last word in lower 10 Mbit	09FFF	0 1001 1111 1111 1111
Instance1	First word in upper 10 Mbit	0A000	0 1010 0000 0000 0000
Instance1	Last word in upper 10 Mbit	13FFF	1 0011 1111 1111 1111

There are separate macro select signals, *inst0_MSN* and *inst1_MSN*, for each instance and separate dataout busses *inst0_DO* and *inst1_DO*, which are multiplexed in the DCU. Apart from these signals both instances share the DRAM output pins of the DCU.

The DRAM Arbitration Unit (DAU) generates a 17 bit address, *dau_dcu_adr[21:5]*, sufficient to address all 256-bit words in the 20 Mbit DRAM. The upper 5 bits are used to select between the two memory instances by gating their MSN pins. If instance1 is selected then the lower 16-bits are translated to map into the 10 Mbit range of that instance. The multiplexing and address translation rules are shown in Table 126.

In the case that the DAU issues a refresh, indicated by *dau_dcu_refresh*, then both macros are selected. The other control signals

Table 126. Instance selection and address translation

dau_dcu_refresh	DAU Address bits dau_dcu_adr[21:17]	Instance selected	inst0_MSN	inst1_MSN	Address translation
0	< 01010	Instance0	MSN	1	A[15:0] = dau_dcu_adr[20:5]
	>= 01010	Instance1	1	MSN	A[15:0] = dau_dcu_adr[21:5] - hA000
1	-	Instance0 and Instance1	MSN	MSN	-

dau_dcu_adr[21:5], *dau_dcu_rwn* and *dau_dcu_cduwpage* are ignored.

The instance selection and address translation logic is shown in Figure 102.

The address translation and instance decode logic also increments the address presented to the DRAM in the case of a page mode write. Pseudo code is given below.

```
if rising_edge(dau_dcu_valid) then
```

```
                    //capture the address from the DAU
                    next_cmdadr[21:5] = dau_dcu_adr[21:5]
                elsif pagemode_adr_inc == 1 then
                    //increment the address
                    next_cmdadr[21:5] = cmdadr[21:5] + 1
                else
                    next_cmdadr[21:5] = cmdadr[21:5]

                if rising_edge(dau_dcu_valid) then
                    //capture the address from the DAU
                    adr_var[21:5] := dau_dcu_adr[21:5]
                else
                    adr_var[21:5] := cmdadr[21:5]

                if adr_var[21:17] < 01010 then
                    //choose instance0
                    instance_sel = 0
                    A[15:0] = adr_var[20:5]
                else
                    //choose instance1
                    instance_sel = 1
                        A[15:0] = adr_var[21:5] - hA000
```

Pseudo code for the select logic, *SEL0*, for DRAM Instance0 is given below.

```
                //instance0 selected or refresh
                if instance_sel == 0 OR dau_dcu_refresh == 1 then
                    inst0_MSN = MSN
                else
                    inst0_MSN = 1
```

Pseudo code for the select logic, *SEL1*, for DRAM Instance1 is given below.

```
                //instance1 selected or refresh
                if instance_sel == 1 OR dau_dcu_refresh == 1 then
                    inst1_MSN = MSN
                else
                    inst1_MSN = 1
```

During a random read, the read data is returned, on *dcu_dau_rdata*, after time T_{acc}, the random access time, which varies between 3 and 8 ns (see Table   ). To avoid any metastability issues the read data must be captured by a flip-flop which is enabled 2 *pclk* cycles or 12.5 ns after the DRAM access has been started. The DCU generates the enable signal *dcu_dau_rvalid* to capture *dcu_dau_rdata*.

The byte write mask *dau_dcu_wmask[31:0]* must be expanded to the bit write mask *bitwritemask[255:0]* needed by the DRAM.

20.14.5   DAU-DCU interface description

The DCU asserts *dcu_dau_adv* in the *MSN2* state to indicate to the DAU to supply the next command. *dcu_dau_adv* causes the DAU to perform arbitration in the *MSN2* cycle. The resulting command is available to the DCU in the following cycle, the *RST* state. The timing is shown in Figure 103. The command to the DRAM must be valid in the *RST* and *MSN1* states, or at least meet the hold time requirement to the *MSN* falling edge at the start of the *MSN1* state.

Note that the DAU issues a valid arbitration result following every *dcu_dau_adv* pulse. If no unit is requesting DRAM access, then a fall-back refresh request will be issued. When *dau_dcu_refresh* is asserted the operation is a refresh and *dau_dcu_adr*, *dau_dcu_rwn* and *dau_dcu_cduwpage* are ignored.

The DCU generates a second signal, *dcu_dau_wadv*, which is asserted in the *RST* state. This indicates to the DAU that it can perform arbitration in advance for non-CPU writes. The reason for performing arbitration in advance for non-CPU writes is explained in " Command Multiplexor Sub-block

Table 136. Command Multiplexor Sub-block IO Definition

Port name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
DIU Read Interface to SoPEC Units			
<unit>_diu_radr[21:5]	17	In	Read address to DIU 17 bits wide (256-bit aligned word).
diu_<unit>_rack	1	Out	Acknowledge from DIU that read request has been accepted and new read address can be placed on *<unit>_diu_radr*
DIU Write Interface to SoPEC Units			
<unit>_diu_wadr[21:5]	17	In	Write address to DIU except CPU, SCB, CDU 17 bits wide (256-bit aligned word)
cpu_diu_wadr[21:4]]	22	In	CPU Write address to DIU (128-bit aligned address.)
cpu_diu_wmask	16	In	Byte enables for CPU write.
cdu_diu_wadr[21:3]	19	In	CDU Write address to DIU 19 bits wide (64-bit aligned word) Addresses cannot cross a 256-bit word DRAM boundary.
diu_<unit>_wack	1	Out	Acknowledge from DIU that write request has been accepted and new write address can be placed on *<unit>_diu_wadr*
Outputs to CPU Interface and Arbitration Logic sub-block			
re_arbitrate	1	Out	Signalling telling the arbitration logic to choose the next

262

			arbitration winner.
re_arbitrate_wadv	1	Out	Signal telling the arbitration logic to choose the next arbitration winner for non-CPU writes 2 timeslots in advance
Debug Outputs to CPU Configuration and Arbitration Logic Sub-block			
write_sel	5	Out	Signal indicating the SoPEC Unit for which the current write transaction is occurring. Encoding is described in Table .
write_complete	1	Out	Signal indicating that write transaction to SoPEC Unit indicated by *write_sel* is complete.
Inputs from CPU Interface and Arbitration Logic sub-block			
arb_gnt	1	In	Signal lasting 1 cycle which indicates arbitration has occurred and *arb_sel* is valid.
arb_sel	5	In	Signal indicating which requesting SoPEC Unit has won arbitration. Encoding is described in Table .
dir_sel	2	In	Signal indicating which sense of access associated with *arb_sel* 00: issue non-CPU write 01: read winner 10: write winner 11: refresh winner
Inputs from Read Write Multiplexor Sub-block			
write_data_valid	2	In	Signal indicating that valid write data is available for the current command. 00=not valid 01=CPU write data valid 10=non-CPU write data valid 11=both CPU and non-CPU write data valid
wdata	256	In	256-bit non-CPU write data
cpu_wdata	32	In	32-bit CPU write data
Outputs to Read Write Multiplexor Sub-block			
write_data_accept	2	Out	Signal indicating the Command Multiplexor has accepted the write data from the write multiplexor 00=not valid 01=accepts CPU write data 10=accepts non-CPU write data 11=not valid
Inputs from DCU			
dcu_dau_adv	1	In	Signal indicating to DAU to supply next command to DCU

dcu_dau_wadv	1	In	Signal indicating to DAU to initiate next non-CPU write
Outputs to DCU			
dau_dcu_adr[21:5]	17	Out	Signal indicating the address for the DRAM access. This is a 256-bit aligned DRAM address.
dau_dcu_rwn	1	Out	Signal indicating the direction for the DRAM access (1=read, 0=write).
dau_dcu_cduwpage	1	Out	Signal indicating if access is a CDU write page mode access (1=CDU page mode, 0=not CDU page mode).
dau_dcu_refresh	1	Out	Signal indicating that a refresh command is to be issued. If asserted *dau_dcu_adr*, *dau_dcu_rwn* and *dau_dcu_cduwpage* are ignored.
dau_dcu_wdata	256	Out	256-bit write data to DCU
dau_dcu_wmask	32	Out	Byte encoded write data mask for 256-bit *dau_dcu_wdata* to DCU

"

The DCU state-machine can stall in the *MSN2* state when the signal *dau_dcu_msn2stall* is asserted by the DAU Arbitration Logic,

The states of the DCU state-machine are summarised in Table 127 .

Table 127. States of the DCU state-machine

State	Description
RST	Restore state
MSN1	Macro select state 1
MSN2	Macro select state 2

20.14.6   DCU state machines

The IBM DRAM has a simple SRAM like interface. The DRAM is accessed as a single bank. The state machine to access the DRAM is shown in Figure 104.

The signal *pagemode_adr_inc* is exported from the DCU as *dcu_dau_cduwaccept*.

dcu_dau_cduwaccept tells the DAU to supply the next write data to the DRAM

20.14.7   CU-11 DRAM timing diagrams

The IBM Cu-11 embedded DRAM datasheet is referenced as [16].

Table 128 shows the timing parameters which must be obeyed for the IBM embedded DRAM.

Table 128. 1.5 V Cu-11 DRAM a.c. parameters

Symbol	Parameter	Min	Max	Units
T_{set}	Input setup to MSN/PGN	1	-	ns
T_{hld}	Input hold to MSN/PGN	2	-	ns
T_{acc}	Random access time	3	8	ns

T_{act}	MSN active time	8	100k	ns
T_{res}	MSN restore time	4	-	ns
T_{cyc}	Random R/W cycle time	12	-	ns
T_{rfc}	Refresh cycle time	12	-	ns
T_{accp}	Page mode access time	1	3.9	ns
T_{pa}	PGN active time	1.6	-	ns
T_{pr}	PGN restore time	1.6	-	ns
T_{pcyc}	PGN cycle time	4	-	ns
T_{mprd}	MSN to PGN restore delay	6	-	ns
T_{actp}	MSN active for page mode	12	-	ns
T_{ref}	Refresh period	-	3.2	ms
T_{pamr}	Page active to MSN restore	4	-	ns

The IBM DRAM is asynchronous. In SoPEC it interfaces to signals clocked on *pclk*. The following timing diagrams show how the timing parameters in Table 129 are satisfied in SoPEC.

20.14.8 Definition of DAU IO

Table 129. DAU interface

5

Port Name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	SoPEC Functional clock
prst_n	1	In	Active-low, synchronous reset in *pclk* domain
dau_dcu_reset_n	1	Out	Active-low, synchronous reset in *pclk* domain. This reset signal, exported to the DCU, incorporates the locally captured DAU version of hard reset (*prst_n*) and the soft reset configuration register bit "*Reset*".
CPU Interface			
cpu_adr	22	In	CPU address bus for both DRAM and configuration register access. 9 bits (bits 10:2) are required to decode the configuration register address space. 22 bits can address the DRAM at byte level. DRAM addresses cannot cross a 256-bit word DRAM boundary.
cpu_dataout	32	In	Shared write data bus from the CPU for DRAM and configuration data
diu_cpu_data	32	Out	Configuration, status and debug read data bus to the CPU
diu_cpu_debug_valid	1	Out	Signal indicating the data on the *diu_cpu_data* bus is valid debug data.

cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_acode	2	In	CPU access code signals. cpu_acode[0] - Program (0) / Data (1) access cpu_acode[1] - User (0) / Supervisor (1) access The DAU will only allow supervisor mode accesses to data space.
cpu_diu_sel	1	In	Block select from the CPU. When *cpu_diu_sel* is high both *cpu_adr* and *cpu_dataout* are valid
diu_cpu_rdy	1	Out	Ready signal to the CPU. When *diu_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the block and for a read cycle this means the data on *diu_cpu_data* is valid.
diu_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
DIU Read Interface to SoPEC Units			
<unit>_diu_rreq	1	In	SoPEC unit requests DRAM read. A read request must be accompanied by a valid read address.
<unit>_diu_radr[21:5]	17	In	Read address to DIU 17 bits wide (256-bit aligned word). Note : "<unit>" refers to non-CPU requesters only. CPU addresses are provided via "cpu_adr".
diu_<unit>_rack	1	Out	Acknowledge from DIU that read request has been accepted and new read address can be placed on *<unit>_diu_radr*
diu_data	64	Out	Data from DIU to SoPEC Units except CPU. First 64-bits is bits 63:0 of 256 bit word Second 64-bits is bits 127:64 of 256 bit word Third 64-bits is bits 191:128 of 256 bit word Fourth 64-bits is bits 255:192 of 256 bit word
dram_cpu_data	256	Out	256-bit data from DRAM to CPU.
diu_<unit>_rvalid	1	Out	Signal from DIU telling SoPEC Unit that valid read data is on the *diu_data* bus
DIU Write Interface to SoPEC Units			
<unit>_diu_wreq	1	In	SoPEC unit requests DRAM write. A write request must be accompanied by a valid write address. Note : "<unit>" refers to non-CPU requesters only.
<unit>_diu_wadr[21:5]	17	In	Write address to DIU except CPU, CDU 17 bits wide (256-bit aligned word)

			Note : "<unit>" refers to non-CPU requesters, excluding the CDU.
scb_diu_wmask[7:0]	8	In	Byte write enables applicable to a given 64-bit quarter-word transferred from the SCB. Note that different mask values are used with each quarter-word. Requirement for the USB host core.
diu_cpu_write_rdy	1	Out	Flag indicating that the CPU posted write buffer is empty.
cpu_diu_wdatavalid	1	In	Write enable for the CPU posted write buffer. Also confirms that the CPU write data, address and mask are valid.
cpu_diu_wdata	128	In	CPU write data which is loaded into the posted write buffer.
cpu_diu_wadr[21:4]	18	In	128-bit aligned CPU write address.
cpu_diu_wmask[15:0]	16	In	Byte enables for 128-bit CPU posted write.
cdu_diu_wadr[21:3]	19	In	CDU Write address to DIU 19 bits wide (64-bit aligned word) Addresses cannot cross a 256-bit word DRAM boundary.
diu_<unit>_wack	1	Out	Acknowledge from DIU that write request has been accepted and new write address can be placed on <unit>_diu_wadr
<unit>_diu_data[63:0]	64	In	Data from SoPEC Unit to DIU except CPU. First 64-bits is bits 63:0 of 256 bit word Second 64-bits is bits 127:64 of 256 bit word . Third 64-bits is bits 191:128 of 256 bit word Fourth 64-bits is bits 255:192 of 256 bit word Note : "<unit>" refers to non-CPU requesters only.
<unit>_diu_wvalid	1	In	Signal from SoPEC Unit indicating that data on <unit>_diu_data is valid. Note : "<unit>" refers to non-CPU requesters only.
Outputs to DCU			
dau_dcu_msn2stall	1	Out	Signal indicating from DAU Arbitration Logic which when de-asserted stalls DCU in MSN2 state.
dau_dcu_adr[21:5]	17	Out	Signal indicating the address for the DRAM access. This is a 256-bit aligned DRAM address.
dau_dcu_rwn	1	Out	Signal indicating the direction for the DRAM access (1=read, 0=write).
dau_dcu_cduwpage	1	Out	Signal indicating if access is a CDU write page mode

			access (1=CDU page mode, 0=not CDU page mode).
dau_dcu_refresh	1	Out	Signal indicating that a refresh command is to be issued. If asserted *dau_dcu_cmd_adr*, *dau_dcu_rwn* and *dau_dcu_cduwpage* are ignored.
dau_dcu_wdata	256	Out	256-bit write data to DCU
dau_dcu_wmask	32	Out	Byte-encoded write data mask for 256-bit *dau_dcu_wdata* to DCU Polarity : A "1" in a bit field of *dau_dcu_wmask* means that the corresponding byte in the 256-bit *dau_dcu_wdata* is written to DRAM.
Inputs from DCU			
dcu_dau_adv	1	In	Signal indicating to DAU to supply next command to DCU
dcu_dau_wadv	1	In	Signal indicating to DAU to initiate next non-CPU write
dcu_dau_refreshcomplete	1	In	Signal indicating that the DCU has completed a refresh.
dcu_dau_rdata	256	In	256-bit read data from DCU.
dcu_dau_rvalid	1	In	Signal indicating valid read data on *dcu_dau_rdata*.

The CPU subsystem bus interface is described in more detail in Section 11.4.3. The DAU block will only allow supervisor-mode accesses to update its configuration registers (i.e. *cpu_acode[1:0]* = b11). All other accesses will result in *diu_cpu_berr* being asserted.

5   20.14.9   DAU Configuration Registers

Table 130. DAU configuration registers

Address (DIU_base +)	Register	#bits	Reset	Description
Reset				
0x00	Reset	1	0x1	A write to this register causes a reset of the DIU. This register can be read to indicate the reset state: 0 - reset in progress 1 - reset not in progress
Refresh				
0x04	RefreshPeriod	9	0x063	Refresh controller. When set to 0 refresh is off, otherwise the value indicates the number of cycles, less one, between each

268

				refresh. [Note that for a system clock frequency of 160MHz, a value exceeding 0x63 (indicating a 100-cycle refresh period) should not be programmed, or the DRAM will malfunction.]
Timeslot allocation and control				
0x08	NumMainTimeslots	6	0x01	Number of main timeslots (1-64) less one
0x0C	CPUPreAccessTimeslots	4	0x0	*(CPUPreAccessTimeslots* + 1) main slots out of a total of (*CPUTotalTimeslots* + 1) are preceded by a CPU access.
0x10	CPUTotalTimeslots	4	0x0	*(CPUPreAccessTimeslots* + 1) main slots out of a total of (*CPUTotalTimeslots* + 1) are preceded by a CPU access.
0x100-0x1FC	MainTimeslot[63:0]	64x4	[63:1][3:0] = 0x0 [0][3:0] = 0xE	Programmable main timeslots (up to 64 main timeslots).
0x200	ReadRoundRobinLevel	12	0x000	For each read requester plus refresh 0 = level1 of round-robin 1 = level2 of round-robin The bit order is defined in Table .
0x204	EnableCPURoundRobin	1	0x1	Allows the CPU to particpate in the unused read round-robin scheme. If disabled, the shared CPU/refresh round-robin position is dedicated solely to refresh.
0x208	RotationSync	1	0x1	Writing 0, followed by 1 to this bit allows the timeslot rotation to advance on a cycle basis which can be determined by the CPU.
0x20C	minNonCPUReadAdr	12	0x800	12 MSBs of lowest DRAM address which may be read by non-CPU requesters.
0x210	minDWUWriteAdr	12	0x800	12 MSBs of lowest DRAM address which may be written to by the DWU.

269

0x214	minNonCPUWriteAdr	12	0x800	12 MSBs of lowest DRAM address which may be written to by non-CPU requesters other than the DWU.
Debug				
0x300	DebugSelect[11:2]	10	0x304	Debug address select. Indicates the address of the register to report on the *diu_cpu_data* bus when it is not otherwise being used. When this signal carries debug information the signal *diu_cpu_debug_valid* will be asserted.
Debug: arbitration and performance				
0x304	ArbitrationHistory	22	-	Bit 0 = arb_gnt Bit 1 = arb_executed Bit 6:2 = arb_sel[4:0] Bit 12:7 = timeslot_number[5:0] Bit 15:13 = access_type[2:0] Bit 16 = back2back_non_cpu_write Bit 17 = sticky_back2back_non_cpu_write (Sticky version of same, cleared on reset.) Bit 18 = rotation_sync Bit 20:19 = rotation_state Bit 21 = sticky_invalid_non_cpu_adr See Section 20.14.9.2 DIU Debug for a description of the fields. Read only register.
0x308	DIUPerformance	31	-	Bit 0 = cpu_diu_rreq Bit 1 = scb_diu_rreq Bit 2 = cdu_diu_rreq Bit 3 = cfu_diu_rreq Bit 4 = lbd_diu_rreq Bit 5 = sfu_diu_rreq Bit 6 = td_diu_rreq Bit 7 = tfs_diu_rreq Bit 8 = hcu_diu_rreq Bit 9 = dnc_diu_rreq Bit 10 = llu_diu_rreq

				Bit 11 = pcu_diu_rreq
				Bit 12 = cpu_diu_wreq
				Bit 13 = scb_diu_wreq
				Bit 14 = cdu_diu_wreq
				Bit 15 = sfu_diu_wreq
				Bit 16 = dwu_diu_wreq
				Bit 17 = refresh_req
				Bit 22:18 = read_sel[4:0]
				Bit 23 = read_complete
				Bit 28:24 = write_sel[4:0]
				Bit 29 = write_complete
				Bit 30 = dcu_dau_refreshcomplete
				See Section 20.14.9.2 DIU Debug for a description of the fields.
				Read only register.
Debug DIU read requesters interface signals				
0x30C	CPUReadInterface	25	-	Bit 0 = cpu_diu_rreq
				Bit 22:1 = cpu_adr[21:0]
				Bit 23 = diu_cpu_rack
				Bit 24 = diu_cpu_rvalid
				Read only register.
0x310	SCBReadInterface	20		Bit 0 = scb_diu_rreq
				Bit 17:1 = scb_diu_radr[21:5]
				Bit 18 = diu_scb_rack
				Bit 19 = diu_scb_rvalid
				Read only register.
0x314	CDUReadInterface	20	-	Bit 0 = cdu_diu_rreq
				Bit 17:1 = cdu_diu_radr[21:5]
				Bit 18 = diu_cdu_rack
				Bit 19 = diu_cdu_rvalid
				Read only register.
0x318	CFUReadInterface	20	-	Bit 0 = cfu_diu_rreq
				Bit 17:1 = cfu_diu_radr[21:5]
				Bit 18 = diu_cfu_rack
				Bit 19 = diu_cfu_rvalid
				Read only register.
0x31C	LBDReadInterface	20	-	Bit 0 = lbd_diu_rreq
				Bit 17:1 = lbd_diu_radr[21:5]
				Bit 18 = diu_lbd_rack

				Bit 19 = diu_lbd_rvalid
				Read only register.
0x320	SFUReadInterface	20	-	Bit 0 = sfu_diu_rreq
				Bit 17:1 = sfu_diu_radr[21:5]
				Bit 18 = diu_sfu_rack
				Bit 19 = diu_sfu_rvalid
				Read only register.
0x324	TDReadInterface	20	-	Bit 0 = td_diu_rreq
				Bit 17:1 = td_diu_radr[21:5]
				Bit 18 = diu_td_rack
				Bit 19 = diu_td_rvalid
				Read only register.
0x328	TFSReadInterface	20	-	Bit 0 = tfs_diu_rreq
				Bit 17:1 = tfs_diu_radr[21:5]
				Bit 18 = diu_tfs_rack
				Bit 19 = diu_tfs_rvalid
				Read only register.
0x32C	HCUReadInterface	20	-	Bit 0 = hcu_diu_rreq
				Bit 17:1 = hcu_diu_radr[21:5]
				Bit 18 = diu_hcu_rack
				Bit 19 = diu_hcu_rvalid
				Read only register.
0x330	DNCReadInterface	20	-	Bit 0 = dnc_diu_rreq
				Bit 17:1 = dnc_diu_radr[21:5]
				Bit 18 = diu_dnc_rack
				Bit 19 = diu_dnc_rvalid
				Read only register.
0x334	LLUReadInterface	20	-	Bit 0 = llu_diu_rreq
				Bit 17:1 = lluu_diu_radr[21:5]
				Bit 18 = diu_llu_rack
				Bit 19 = diu_llu_rvalid
				Read only register.
0x338	PCUReadInterface	20	-	Bit 0 = pcu_diu_rreq
				Bit 17:1 = pcu_diu_radr[21:5]
				Bit 18 = diu_pcu_rack
				Bit 19 = diu_pcu_rvalid
				Read only register.
Debug DIU write requesters interface signals				
0x33C	CPUWriteInterface	27	-	Bit 0 = cpu_diu_wreq

				Bit 22:1 = cpu_adr[21:0]
				Bit 24:23 = cpu_diu_wmask[1:0]
				Bit 25 = diu_cpu_wack
				Bit 26 = cpu_diu_wvalid
				Read only register.
0x340	SCBWriteInterface	20	-	Bit 0 = scb_diu_wreq
				Bit 17:1 = scb_diu_wadr[21:5]
				Bit 18 = diu_scb_wack
				Bit 19 = scb_diu_wvalid
				Read only register.
0x344	CDUWriteInterface	22	-	Bit 0 = cdu_diu_wreq
				Bit 19:1 = cdu_diu_wadr[21:3]
				Bit 20 = diu_cdu_wack
				Bit 21 = cdu_diu_wvalid
				Read only register.
0x348	SFUWriteInterface	20	-	Bit 0 = sfu_diu_wreq
				Bit 17:1 = sfu_diu_wadr[21:5]
				Bit 18 = diu_sfu_wack
				Bit 19 = sfu_diu_wvalid
				Read only register.
0x34C	DWUWriteInterface	20	-	Bit 0 = dwu_diu_wreq
				Bit 17:1 = dwu_diu_wadr[21:5]
				Bit 18 = diu_dwu_wack
				Bit 19 = dwu_diu_wvalid
				Read only register.
Debug DAU-DCU interface signals				
0x350	DAU-DCUInterface	25	-	Bit 16:0 = dau_dcu_adr[21:5]
				Bit 17 = dau_dcu_rwn
				Bit 18 = dau_dcu_cduwpage
				Bit 19 = dau_dcu_refresh
				Bit 20 = dau_dcu_msn2stall
				Bit 21 = dcu_dau_adv
				Bit 22 = dcu_dau_wadv
				Bit 23 = dcu_dau_refreshcomplete
				Bit 24 = dcu_dau_rvalid
				Read only register.

Each main timeslot can be assigned a SoPEC DIU requestor according to Table 131.

Table 131. SoPEC DIU requester encoding for main timeslots.

Name	Index (binary)	Index (HEX)
Write		
SCB(W)	b0_0000	0x00
CDU(W)	b0001	0x1
SFU(W)	b0010	0x2
DWU	b0011	0x3
Read		
SCB(R)	b0100	0x4
CDU(R)	b0101	0x5
CFU	b0110	0x6
LBD	b0111	0x7
SFU(R)	b1000	0x8
TE(TD)	b1001	0x9
TE(TFS)	b1010	0xA
HCU	b1011	0xB
DNC	b1100	0xC
LLU	b1101	0xD
PCU	b1110	0xE

ReadRoundRobinLevel and *ReadRoundRobinEnable* registers are encoded in the bit order defined in Table 132.

Table 132. Read round-robin registers bit order

Name	Bit index
SCB(R)	0
CDU(R)	1
CFU	2
LBD	3
SFU(R)	4
TE(TD)	5
TE(TFS)	6
HCU	7
DNC	8
LLU	9
PCU	10
CPU Refresh	11

5   *20.14.9.1 Configuration register reset state*

The *RefreshPeriod* configuration register has a reset value of 0x063 which ensures that a refresh will occur every 100 cycles and the contents of the DRAM will remain valid.

The *CPUPreAccessTimeslots* and *CPUTotalTimeslots* configuration registers both have a reset value of 0x0. Matching values in these two registers means that *every* slot has a CPU pre-acess. *NumMainTimeslots* is reset to 0x1, so there are just 2 main timeslots in the rotation initially. These slots alternate between SCB writes and PCU reads, as defined by the reset value of

5   *MainTimeslot[63:0]*, thus respecting at reset time the general rule that adjacent non-CPU writes are not permitted.

The first access issued by the DIU after reset will be a refresh.

20.14.9.2 DIU Debug

External visibility of the DIU must be provided for debug purposes. To facilitate this debug

10   registers are added to the DIU address space.

The DIU CPU system data bus *diu_cpu_data[31:0]* returns configuration and status register information to the CPU. When a configuration or status register is not being read by the CPU debug data is returned on *diu_cpu_data[31:0]* instead. An accompanying active high *diu_cpu_debug_valid* signal is used to indicate when the data bus contains valid debug data.

15   The DIU features a *DebugSelect* register that controls a local multiplexor to determine which register is output on *diu_cpu_data[31:0]*.

Three kinds of debug information are gathered:

a. The order and access type of DIU requesters winning arbitration.

This information can be obtained by observing the signals in the *ArbitrationHistory* debug register

20   at *DIU_Base+0x304* described in Table 133.

Table 133. *ArbitrationHistory* debug register description, DIU_base+0x304

Field name	Bits	Description
arb_gnt	1	Signal lasting 1 cycle which is asserted in the cycle following a main arbitration or pre-arbitration.
arb_executed	1	Signal lasting 1 cycle which indicates that an arbitration result has actually been executed. Is used to differentiate between *pre*-arbitration and *main* arbitration, both of which cause *arb_gnt* to be asserted. If *arb_executed* and *arb_gnt* are both high, then a main (executed) arbitration is indicated.
arb_sel	5	Signal indicating which requesting SoPEC Unit has won arbitration. Encoding is described in Table . *Refresh* winning arbitration is indicated by *access_type*.
timeslot_number	6	Signal indicating which main timeslot is either currently being serviced, or about to be serviced. The latter case applies where a main slot is pre-empted by a CPU pre-access or a scheduled refresh.
access_type	3	Signal indicating the origin of the winning arbitration 000 = Standard CPU pre-access. 001 = Scheduled refresh.

		010 = Standard non-CPU timeslot. 011 = CPU access via unused read/write slot, re-allocated by round robin. 100 = Non-CPU write via unused write slot, re-allocated at pre-arbitration. 101 = Non-CPU read via unused read/write slot, re-allocated by round robin. 110 = Refresh via unused read/write slot, re-allocated by round robin. 111 = CPU / Refresh access due to *RotationSync* = 0.
back2back_non_c pu_write	1	Instantaneous indicator of attempted illegal back-to-back non-CPU write. (Recall from section 20.7.2.3 on page 212 that the second write of any such pair is disregarded and re-allocated via the unused read round-robin scheme.)
sticky_back2back_ non_cpu_write	1	Sticky version of same, cleared on reset.
rotation_sync	1	Current value of the *RotationSync* configuration bit.
rotation_state	2	These bits indicate the current status of pre-arbitation and main timeslot rotation, as a result of the *RotationSync* setting. 00 = Pre-arb enabled, rotation enabled. 01 = Pre-arb disabled, rotation enabled. 10 = Pre-arb disabled, rotation disabled. 11 = Pre-arb enabled, rotation disabled. 00 is the normal functional setting when *RotationSync* is 1. 01 indicates that pre-arbitration has halted at the end of its rotation because of *RotationSync* having been cleared. However the main arbitration has yet to finish *its* current rotation. 10 indicates that both pre-arb and the main rotation have halted, due to *RotationSync* being 0 and that only CPU accesses and refreshes are allowed. 11 indicates that *RotationSync* has just been changed from 0 to 1 and that pre-arbitration is being given a head start to look ahead for non-CPU writes, in advance of the main rotation starting up again.
sticky_invalid_non _cpu_adr	1	Sticky bit to indicate an attempted non-CPU access with an invalid address. Cleared by reset or by an explicit write by the CPU.

Table 134. *arb_sel, read_sel* and *write_sel* encoding

Name	Index (binary)	Index (HEX)
Write		
SCB(W)	b0_0000	0x00
CDU(W)	b0_0001	0x01
SFU(W)	b0_0010	0x02
DWU	b0_0011	0x03
Read		
SCB(R)	b0_0100	0x04
CDU(R)	b0_0101	0x05
CFU	b0_0110	0x06
LBD	b0_0111	0x07
SFU(R)	b0_1000	0x08
TE(TD)	b0_1001	0x09
TE(TFS)	b0_1010	0x0A
HCU	b0_1011	0x0B
DNC	b0_1100	0x0C
LLU	b0_1101	0x0D
PCU	b0_1110	0x0E
Refresh		
Refresh	b0_1111	0x0F
CPU		
CPU(R)	b1_0000	0x10
CPU(W)	b1_0001	0x11

The encoding for *arb_sel* is described in Table 134.

b. The time between a DIU requester requesting an access and completing the access.

This information can be obtained by observing the signals in the *DIUPerformance* debug register

at *DIU_Base+0x308* described in Table 135. The encoding for *read_sel* and *write_sel* is described in Table . The data collected from *DIUPerformance* can be post-processed to count the number of cycles between a unit requesting DIU access and the access being completed.

Table 135. *DIUPerformance* debug register description, DIU_base+0x308

Field name	Bits	Description
<unit>_diu_rreq	12	Signal indicating that SoPEC unit requests DRAM read.
<unit>_diu_wreq	5	Signal indicating that SoPEC unit requests DRAM write.
refresh_req	1	Signal indicating that *refresh* has requested a DIU access.
read_sel[4:0]	5	Signal indicating the SoPEC Unit for which the current read

		transaction is occurring. Encoding is described in Table .
read_complete	1	Signal indicating that read transaction to SoPEC Unit indicated by *read_sel* is complete i.e. that the last read data has been output by the DIU.
write_sel[4:0]	5	Signal indicating the SoPEC Unit for which the current write transaction is occurring. Encoding is described in Table .
write_complete	1	Signal indicating that write transaction to SoPEC Unit indicated by *write_sel* is complete i.e. that the last write data has been transferred to the DIU. .
dcu_refresh_complete	1	Signal indicating that *refresh* has completed.

c. Interface signals to DIU requestors and DAU-DCU interface.

All interface signals with the exception of data busses at the interfaces between the DAU and DCU and DIU write and read requestors can be monitored in debug mode by observing debug registers *DIU_Base+0x314* to *DIU_Base+0x354*.

20.14.10   DRAM Arbitration Unit (DAU)

The DAU is shown in Figure 101.

The DAU is composed of the following sub-blocks

a.   CPU Configuration and Arbitration Logic sub-block.

b.   Command Multiplexor sub-block.

c.   Read and Write Data Multiplexor sub-block.

The function of the DAU is to supply DRAM commands to the DCU.

- The DCU requests a command from the DAU by asserting *dcu_dau_adv.*

- ' The DAU Command Multiplexor requests the Arbitration Logic sub-block to arbitrate the next DRAM access. The Command Multiplexor passes *dcu_dau_adv* as the *re_arbitrate* signal to the Arbitration Logic sub-block.

- If the *RotationSync* bit has been cleared, then the arbitration logic grants exclusive access to the CPU and scheduled refreshes. If the bit has been set, regular arbitration occurs. A detailed description of *RotationSync* is given in section 20.14.12.2.1 on page 295.

- Until the Arbitration Logic has a valid result it stalls the DCU by asserting *dau_dcu_msn2stall*. The Arbitration Logic then returns the selected arbitration winner to the Command Multiplexor which issues the command to the DRAM. The Arbitration Logic could stall for example if it selected a shared read bus access but the Read Multiplexor indicated it was busy by de-asserting *read_cmd_rdy[1]*.

- In the case of a read command the read data from the DRAM is multiplexed back to the read requestor by the Read Multiplexor. In the case of a write operation the Write Multiplexor multiplexes the write data from the selected DIU write requestor to the DCU before the write command can occur. If the write data is not available then the Command Multiplexor will keep *dau_dcu_valid* de-asserted. This will stall the DCU until the write command is ready to be issued.

- Arbitration for non-CPU writes occurs in advance. The DCU provides a signal *dcu_dau_wadv* which the Command Multiplexor issues to the Arbitrate Logic as *re_arbitrate_wadv*. If arbitration is blocked by the Write Multiplexor being busy, as indicated by *write_cmd_rdy[1]* being de-asserted, then the Arbitration Logic will stall the DCU by asserting *dau_dcu_msn2stall* until the Write Multiplexor is ready.

20.14.10.1   Read Accesses

The timing of a non-CPU DIU read access are shown in Figure 109. Note *re_arbitrate* is asserted in the *MSN2* state of the previous access.

Note the *fixed* timing relationship between the read acknowledgment and the first rvalid for all non-CPU reads. This means that the second and any later reads in a back-to-back non-CPU sequence have their acknowledgments asserted one cycle later, i.e. in the "MSN1" DCU state.

The timing of a CPU DIU read access is shown in Figure 110. Note *re_arbitrate* is asserted in the *MSN2* state of the previous access.

Some points can be noted from Figure 109 and Figure 110.

DIU requests:

- For non-CPU accesses the *<unit>_diu_rreq* signals are registered before the arbitration can occur.

- For CPU accesses the *cpu_diu_rreq* signal is not registered to reduce CPU DIU access latency.

Arbitration occurs when the *dcu_dau_adv* signal from the DCU is asserted. The DRAM address for the arbitration winner is available in the next cycle, the *RST* state of the DCU.

The DRAM access starts in the *MSN1* state of the DCU and completes in the *RST* state of the DCU.

Read data is available:

- In the *MSN2* cycle where it is output unregistered to the CPU

- In the *MSN2* cycle and registered in the DAU before being output in the next cycle to all other read requestors in order to ease timing.

The DIU protocol is in fact:

- Pipelined i.e. the following transaction is initiated while the previous transfer is in progress.

- Split transaction i.e. the transaction is split into independent address and data transfers.

Some general points should be noted in the case of CPU accesses:

- Since the CPU request is not registered in the DIU before arbitration, then the CPU must generate the request, route it to the DAU and complete arbitration all in 1 cycle. To facilitate this CPU access is arbitrated late in the arbitration cycle (see Section 20.14.12.2).

- Since the CPU read data is not registered in the DAU and CPU read data is available 8 ns after the start of the access then 4.5 ns are available for routing and any shallow logic before the CPU read data is captured by the CPU (see Section 20.14.4).

The phases of CPU DIU read access are shown in Figure 111. This matches the timing shown in Table 135.

20.14.10.2   Write Accesses

CPU writes are posted into a 1-deep write buffer in the DIU and written to DRAM as shown below in Figure 112.

The sequence of events is as follows :-

- [1] The DIU signals that its buffer for CPU posted writes is empty (and has been for some time in the case shown).

- [2] The CPU asserts "cpu_diu_wdatavalid" to enable a write to the DIU buffer and presents valid address, data and write mask. The CPU considers the write posted and thus complete in the cycle following [2] in the diagram below.

- [3] The DIU stores the address/data/mask in its buffer and indicates to the arbitration logic that a posted write wishes to participate in any upcoming arbitration.

- [4] Provided the CPU still has a pre-access entitlement left, or is next in line for a round-robin award, a slot is arbitrated in favour of the posted write. Note that posted CPU writes have higher arbitration priority than simultaneous CPU reads.

- [5] The DRAM write occurs.

- [6] The earliest that "diu_cpu_write_rdy" can be re-asserted in the "MSN1" state of the DRAM write. In the same cycle, having seen the re-assertion, the CPU can asynchronously turn around "cpu_diu_wdatavalid" and enable a subsequent posted write, should it wish to do so.

The timing of a non-CPU/non-CDU DIU write access is shown below in Figure 113.

Compared to a read access, write data is only available from the requester 4 cycles after the address. An extra cycle is used to ensure that data is first registered in the DAU, before being despatched to DRAM. As a result, writes are pre-arbitrated 5 cycles in advance of the main arbitration decision to actually write the data to memory.

The diagram above shows the following sequence of events :-

- [1] A non-CPU block signals a write request.

- [2] A registered version of this is available to the DAU arbitration logic.

- [3] Write pre-arbitration occurs in favour of the requester.

- [4] A write acknowledgment is returned by the DIU.

- [5] The pre-arbitration will only be upheld if the requester supplies 4 consecutive write data quarter-words, qualified by an asserted wvalid flag.

- [6] Provided this has happened, the main arbitration logic is in a position at [6] to reconfirm the pre-arbitration decision. Note however that such reconfirmation may have to wait a further one or two DRAM accesses, if the write is pre-empted by a CPU pre-access and/or a scheduled refresh.

- [7] This is the *earliest* that the write to DRAM can occur.

- Note that neither the arbitration at [8] nor the pre-arbitration at [9] can award its respective slot to a non-CPU write, due to the ban on back-to-back accesses.

The timing of a CDU DIU write access is shown overleaf in Figure 114.

This is simular to a regular non-CPU write access, but uses page mode to carry out 4 consecutive DRAM writes to contiguous addresses. As a consequence, subsequent accesses are delayed by

6 cycles, as shown in the diagram. Note that a new write can be pre-arbitrated at [10] in Figure 114.

20.14.11 Command Multiplexor Sub-block

Table 136. Command Multiplexor Sub-block IO Definition

Port name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
DIU Read Interface to SoPEC Units			
<unit>_diu_radr[21:5]	17	In	Read address to DIU 17 bits wide (256-bit aligned word).
diu_<unit>_rack	1	Out	Acknowledge from DIU that read request has been accepted and new read address can be placed on *<unit>_diu_radr*
DIU Write Interface to SoPEC Units			
<unit>_diu_wadr[21:5]	17	In	Write address to DIU except CPU, SCB, CDU 17 bits wide (256-bit aligned word)
cpu_diu_wadr[21:4]]	22	In	CPU Write address to DIU (128-bit aligned address.)
cpu_diu_wmask	16	In	Byte enables for CPU write.
cdu_diu_wadr[21:3]	19	In	CDU Write address to DIU 19 bits wide (64-bit aligned word) Addresses cannot cross a 256-bit word DRAM boundary.
diu_<unit>_wack	1	Out	Acknowledge from DIU that write request has been accepted and new write address can be placed on *<unit>_diu_wadr*
Outputs to CPU Interface and Arbitration Logic sub-block			
re_arbitrate	1	Out	Signalling telling the arbitration logic to choose the next arbitration winner.
re_arbitrate_wadv	1	Out	Signal telling the arbitration logic to choose the next arbitration winner for non-CPU writes 2 timeslots in advance
Debug Outputs to CPU Configuration and Arbitration Logic Sub-block			
write_sel	5	Out	Signal indicating the SoPEC Unit for which the current write transaction is occurring. Encoding is described in Table .
write_complete	1	Out	Signal indicating that write transaction to SoPEC Unit indi-

			cated by *write_sel* is complete.
Inputs from CPU Interface and Arbitration Logic sub-block			
arb_gnt	1	In	Signal lasting 1 cycle which indicates arbitration has occurred and *arb_sel* is valid.
arb_sel	5	In	Signal indicating which requesting SoPEC Unit has won arbitration. Encoding is described in Table .
dir_sel	2	In	Signal indicating which sense of access associated with *arb_sel* 00: issue non-CPU write 01: read winner 10: write winner 11: refresh winner
Inputs from Read Write Multiplexor Sub-block			
write_data_valid	2	In	Signal indicating that valid write data is available for the current command. 00=not valid 01=CPU write data valid 10=non-CPU write data valid 11=both CPU and non-CPU write data valid
wdata	256	In	256-bit non-CPU write data
cpu_wdata	32	In	32-bit CPU write data
Outputs to Read Write Multiplexor Sub-block			
write_data_accept	2	Out	Signal indicating the Command Multiplexor has accepted the write data from the write multiplexor 00=not valid 01=accepts CPU write data 10=accepts non-CPU write data 11=not valid
Inputs from DCU			
dcu_dau_adv	1	In	Signal indicating to DAU to supply next command to DCU
dcu_dau_wadv	1	In	Signal indicating to DAU to initiate next non-CPU write
Outputs to DCU			
dau_dcu_adr[21:5]	17	Out	Signal indicating the address for the DRAM access. This is a 256-bit aligned DRAM address.
dau_dcu_rwn	1	Out	Signal indicating the direction for the DRAM access (1=read, 0=write).
dau_dcu_cduwpage	1	Out	Signal indicating if access is a CDU write page mode access (1=CDU page mode, 0=not CDU page mode).
dau_dcu_refresh	1	Out	Signal indicating that a refresh command is to be issued. If

			asserted *dau_dcu_adr, dau_dcu_rwn* and *dau_dcu_cduwpage* are ignored.
dau_dcu_wdata	256	Out	256-bit write data to DCU
dau_dcu_wmask	32	Out	Byte encoded write data mask for 256-bit *dau_dcu_wdata* to DCU

20.14.11.1   Command Multiplexor Sub-block Description

The Command Multiplexor sub-block issues read, write or refresh commands to the DCU, according to the SoPEC Unit selected for DRAM access by the Arbitration Logic. The Command Multiplexor signals the Arbitration Logic to perform arbitration to select the next SoPEC Unit for DRAM access. It does this by asserting the *re_arbitrate* signal. *re_arbitrate* is asserted when the DCU indicates on *dcu_dau_adv* that it needs the next command.

The Command Multiplexor is shown in Figure 115.

Initially, the issuing of commands is described. Then the additional complexity of handling non-CPU write commands arbitrated in advance is introduced.

DAU-DCU interface

See Section 20.14.5 for a description of the DAU-DCU interface.

Generating re_arbitrate

The condition for asserting *re_arbitrate* is that the DCU is looking for another command from the DAU. This is indicated by *dcu_dau_adv* being asserted.

$$re\_arbitrate = dcu\_dau\_adv$$

Interface to SoPEC DIU requestors

When the Command Multiplexor initiates arbitration by asserting *re_arbitrate* to the Arbitration Logic sub-block, the arbitration winner is indicated by the *arb_sel[4:0]* and *dir_sel[1:0]* signals returned from the Arbitration Logic. The validity of these signals is indicated by *arb_gnt*. The encoding of *arb_sel[4:0]* is shown in Table .

The value of *arb_sel[4:0]* is used to control the *steering multiplexor* to select the DIU address of the winning arbitration requestor. The *arb_gnt* signal is decoded as an acknowledge, *diu_<unit>_*ack* back to the winning DIU requestor. The timing of these operations is shown in Figure 116. *adr[21:0]* is the output of the steering multiplexor controlled by *arb_sel[4:0]*. The steering multiplexor can acknowledge DIU requestors in successive cycles.

Command Issuing Logic

The address presented by the winning SoPEC requestor from the steering multiplexor is presented to the command issuing logic together with *arb_sel[4:0]* and *dir_sel[1:0]*.

The command issuing logic translates the winning command into the signals required by the DCU. *adr_[21:0]*, *arb_sel[4:0]* and *dir_sel[1:0]* comes from the steering multiplexor.

$$dau\_dcu\_adr[21:5] = adr[21:5]$$

283

```
          dau_dcu_rwn = (dir_sel[1:0] == read)
          dau_dcu_cduwpage = (arb_sel[4:0] == CDU write)
             dau_dcu_refresh = (dir_sel[1:0]== refresh)
```

5   *dau_dcu_valid* indicates that a valid command is available to the DCU.

For a write command, *dau_dcu_valid* will not be asserted until there is also valid write data present. This is indicated by the signal *write_data_valid[1:0]* from the Read Write Data Multiplexor sub-block.

For a write command, the data issued to the DCU on *dau_dcu_wdata[255:0]* is multiplexed from

10 *cpu_wdata[31:0]* and *wdata[255:0]* depending on whether the write is a CPU or non-CPU write. The write data from the Write Multiplexor for the CDU is available on *wdata[63:0]*. This data must be issued to the DCU on *dau_dcu_wdata[255:0]*. *wdata[63:0]* is copied to each 64-bit word of *dau_dcu_wdata[255:0]*.

15
```
          dau_dcu_wdata[255:0] = 0x00000000
          if (arb_sel[4:0]==CPU write) then
               dau_dcu_wdata[31:0] = cpu_wdata[31:0]
          elsif (arb_sel[4:0]==CDU write)) then
               dau_dcu_wdata[63:0] = wdata[63:0]
20             dau_dcu_wdata[127:64] = wdata[63:0]
               dau_dcu_wdata[191:128] = wdata[63:0]
               dau_dcu_wdata[255:192] = wdata[63:0]
          else
               dau_dcu_wdata[255:0] = wdata[255:0]
```
25

CPU write masking

The CPU write data bus is only 128 bits wide. *cpu_diu_wmask[15:0]* indicates how many bytes of that 128 bits should be written. The associated address *cpu_diu_wadr[21:4]* is a 128-bit aligned address. The actual DRAM write must be a 256-bit access. The command multiplexor issues the

30 256-bit DRAM address to the DCU on *dau_dcu_adr[21:5]*. *cpu_diu wadr[4]* and *cpu_diu_wmask[15:0]* are used jointly to construct a byte write mask *dau_dcu_wmask[31:0]* for this 256-bit write access.

CDU write masking

The CPU performs four 64-bit word writes to 4 contiguous 256-bit DRAM addresses with the first

35 address specified by *cdu_diu_wadr[21:3]*. The write address *cdu_diu_wadr[21:5]* is 256-bit aligned with bits *cdu_diu_wadr[4:3]* allowing the 64-bit word to be selected. If these 4 DRAM words lie in the same DRAM row then an efficient access will be obtained.

The command multiplexor logic must issue 4 successive accesses to 256-bit DRAM addresses *cdu_diu_wadr[21:5],+1,+2,+3*.

40 *dau_dcu_wmask[31:0]* indicates which 8 bytes (64-bits) of the 256-bit word are to be written. *dau_dcu_wmask[31:0]* is calculated using *cdu_diu_wadr[4:3]* i.e. bits 8**cdu_diu_wadr[4:3]* to 8*(*cdu_diu_wadr[4:3]+1)-1* of *dau_dcu_wmask[31:0]*are asserted.

Arbitrating non-CPU writes in advance

In the case of a non-CPU write commands, the write data must be transferred from the SoPEC requester before the write can occur. Arbitration should occur early to allow for any delay for the write data to be transferred to the DRAM.

5   Figure 113 indicates that write data transfer over 64-bit busses will take a further 4 cycles after the address is transferred. The arbitration must therefore occur 4 cycles in advance of arbitration for read accesses, Figure 109 and Figure 110, or for CPU writes Figure 112. Arbitration of CDU write accesses, Figure 114, should take place 1 cycle in advance of arbitration for read and CPU write accesses. To simplify implementation CDU write accesses are arbitrated 4 cycles in advance,

10   similar to other non-CPU writes.

The Command Multiplexor generates another version of *re_arbitrate* called *re_arbitrate_wadv* based on the signal *dcu_dau_wadv* from the DCU. In the 3 cycle DRAM access *dcu_dau_adv* and therefore *re_arbitrate* are asserted in the *MSN2* state of the DCU state-machine. *dcu_dau_wadv* and therefore *re_arbitrate_wadv* will therefore be asserted in the following *RST* state, see Figure

15   117. This matches the timing required for non-CPU writes shown in Figure 113 and Figure 114.

            re_arbitrate_wadv causes the Arbitration Logic to perform an arbitration for non-
            CPU in advance.

20
                    re_arbitrate = dcu_dau_adv
                    re_arbitrate_wadv = dcu_dau_wadv

If the winner of this arbitration is a non-CPU write then *arb_gnt* is asserted and the arbitration winner is output on *arb_sel[4:0]* and *dir_sel[1:0]*. Otherwise *arb_gnt* is not asserted.

25   Since non-CPU write commands are arbitrated early, the non-CPU command is not issued to the DCU immediately but instead written into an advance command register.

                    if (arb_sel(4:0 == non-CPU write) then
                        advance_cmd_register[3:0] = arb_sel[4:0]
30   advance_cmd_register[5:4] = dir_sel[1:0]
                        advance_cmd_register[27:6] = adr[21:0]

If a DCU command is in progress then the arbitration in advance of a non-CPU write command will overwrite the steering multiplexor input to the command issuing logic. The arbitration in

35   advance happens in the DCU *MSN1* state. The new command is available at the steering multiplexor in the *MSN2* state. The   command in progress will have been latched in the DRAM by *MSN* falling at the start of the *MSN1* state.

Issuing non-CPU write commands

40   The *arb_sel[4:0]* and *dir_sel[1:0]* values generated by the Arbitration Logic reflect the *out of order* arbitration sequence.

This out of order arbitration sequence is exported to the Read Write Data Multiplexor sub-block. This is so that write data in available in time for the actual write operation to DRAM. Otherwise a latency would be introduced every time a write command is selected.

However, the Command Multiplexor must execute the command stream *in-order*.

5   In-order command execution is achieved by waiting until *re_arbitrate* has advanced to the non-CPU write timeslot from which *re_arbitrate_wadv* has previously issued a non-CPU write written to the advance command register.

If *re_arbitrate_wadv* arbitrates a non-CPU write in advance then within the Arbitration Logic the timeslot is marked to indicate whether a write was issued.

10   When *re_arbitrate* advances to a write timeslot in the Arbitration Logic then one of two actions can occur depending on whether the slot was marked by *re_arbitrate_wadv* to indicate whether a write was issued or not.

•   Non-CPU write arbitrated by *re_arbitrate_wadv*

If the timeslot has been marked as having issued a write then the arbitration logic responds to

15   *re_arbitrate* by issuing *arb_sel[4:0]*, *dir_sel[1:0]* and asserting *arb_gnt* as for a normal arbitration but selecting a non-CPU write access.   Normally, *re_arbitrate* does not issue non-CPU write accesses. Non-CPU writes are arbitrated by *re_arbitrate_wadv*. *dir_sel[1:0] == 00* indicates a non-CPU write issued by *re_arbitrate*.

The command multiplexor does not write the command into the advance command register as it

20   has already been placed there earlier by *re_arbitrate_wadv*. Instead, the already present write command in the advance command register is issued when *write_data_valid[1]* = 1. Note, that the value of *arb_sel[4:0]* issued by *re_arbitrate* could specify a different write than that in the advance command register since time has advanced. It is always the command in the advance command register that is issued. The steering multiplexor in this case must not issue an acknowledge back

25   to SoPEC requester indicated by the value of *arb_sel[4:0]*.

```
            if (dir_sel[1:0] == 00) then
                command_issuing_logic[27:0]                    ==
        advance_cmd_register[27:0]
30          else
                command_issuing_logic[27:0]                    ==
        steering_multiplexor[27:0]


            ack = arb_gnt AND NOT (dir_sel[1:0] == 00)
35
```

•   Non-CPU write not arbitrated by *re_arbitrate_wadv*

If the timeslot has been marked as not having issued a write, the *re_arbitrate* will use the un-used read timeslot selection to replace the un-used write timeslot with a read timeslot according to Section 20.10.6.2 Unused read timeslots allocation.

40   The mechanism for write timeslot arbitration selects non-CPU writes in advance. But the selected non-CPU write is stored in the Command Multiplexor and issued when the write data is available.

This means that even if this timeslot is overwritten by the CPU reprogramming the timeslot before the write command is actually issued to the DRAM, the originally arbitrated non-CPU write will always be correctly issued.

5   Accepting write commands

When a write command is issued then *write_data_accept[1:0]* is asserted. This tells the Write Multiplexor that the current write data has been accepted by the DRAM and the write multiplexor can receive write data from the next arbitration winner if it is a write. *write_data_accept[1:0]* differentiates between CPU and
10   non-CPU writes. A write command is known to have been issued when *re_arbitrate_wadv* to decide on the next command is detected.

In the case of CDU writes the DCU will generate a signal *dcu_dau_cduwaccept* which tells the Command Multiplexor to issue a *write_data_accept[1]*. This will result in the Write Multiplexor supplying the next CDU write data to the DRAM.

15

```
write_data_accept[0] = RISING EDGE(re_arbitrate_wadv)
                                                AND
command_issuing_logic(dir_sel[1]==1)
                                                AND
command_issuing_logic(arb_sel[4:0]==CPU)


write_data_accept[1] = (RISING EDGE(re_arbitrate_wadv)
                                                AND
command_issuing_logic(dir_sel[1]==1)
                                                AND
command_issuing_logic(arb_sel[4:0]==non_CPU))
                                                OR
dcu_dau_cduwaccept==1
```

30   Debug logic output to CPU Configuration and Arbitration Logic sub-block
write_sel[4:0] reflects the value of *arb_sel[4:0]* at the command issuing logic. The signal *write_complete* is asserted when every any bit of *write_data_accept[1:0]* is asserted.

```
write_complete      =      write_data_accept[0]      OR
35   write_data_accept[0]
```

write_sel[4:0] and *write_complete* are CPU readable from the *DIUPerformance* and *WritePerformance* status registers. When *write_complete* is asserted *write_sel[4:0]* will indicate which write access the DAU has issued.

40   20.14.12 CPU Configuration and Arbitration Logic Sub-block
        Table 137. CPU Configuration and Arbitration Logic Sub-block IO Definition

Port name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
CPU Interface data and control signals			
cpu_adr[10:2]	9	In	9 bits (bits 10:2) are required to decode the configuration register address space.
cpu_dataout	32	In	Shared write data bus from the CPU for DRAM and configuration data
diu_cpu_data	32	Out	Configuration, status and debug read data bus to the CPU
diu_cpu_debug_valid	1	Out	Signal indicating the data on the *diu_cpu_data* bus is valid debug data.
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_acode	2	In	CPU access code signals. cpu_acode[0] - Program (0) / Data (1) access cpu_acode[1] - User (0) / Supervisor (1) access The DAU will only allow supervisor mode accesses to data space.
cpu_diu_sel	1	In	Block select from the CPU. When *cpu_diu_sel* is high both *cpu_adr* and *cpu_dataout* are valid
diu_cpu_rdy	1	Out	Ready signal to the CPU. When *diu_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the block and for a read cycle this means the data on *diu_cpu_data* is valid.
diu_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
DIU Read Interface to SoPEC Units			
<unit>_diu_rreq	11	In	SoPEC unit requests DRAM read.
DIU Write Interface to SoPEC Units			
diu_cpu_write_rdy	1	In	Indicator that CPU posted write buffer is empty.
<unit>_diu_wreq	4	In	Non- CPU SoPEC unit requests DRAM write.
Inputs from Command Multiplexor sub-block			
re_arbitrate	1	In	Signal telling the arbitration logic to choose the next arbitration winner.
re_arbitrate_wadv	1	In	Signal telling the arbitration logic to choose the next arbitration winner for non-CPU writes 2 timeslots in advance

Outputs to DCU			
dau_dcu_msn2stall	1	Out	Signal indicating from DAU Arbitration Logic which when asserted stalls DCU in *MSN2* state.
Inputs from Read and Write Multiplexor sub-block			
read_cmd_rdy	2	In	Signal indicating that read multiplexor is ready for next read read command. 00=not ready 01=ready for CPU read 10=ready for non-CPU read 11=ready for both CPU and non-CPU reads
write_cmd_rdy	2	In	Signal indicating that write multiplexor is ready for next write command. 00=not ready 01=ready for CPU write 10=ready for non-CPU write 11=ready for both CPU and non-CPU write
Outputs to other DAU sub-block s			
arb_gnt	1	In	Signal lasting 1 cycle which indicates arbitration has occurred and *arb_sel* is valid.
arb_sel	5	In	Signal indicating which requesting SoPEC Unit has won arbitration. Encoding is described in Table .
dir_sel	2	In	Signal indicating which sense of access associated with *arb_sel* 00: issue non-CPU write 01: read winner 10: write winner 11: refresh winner
Debug Inputs from Read-Write Multiplexor sub-block			
read_sel	5	In	Signal indicating the SoPEC Unit for which the current read transaction is occurring. Encoding is described in Table .
read_complete	1	In	Signal indicating that read transaction to SoPEC Unit indicated by *read_sel* is complete.
Debug Inputs from Command Multiplexor sub-block			
write_sel	5	In	Signal indicating the SoPEC Unit for which the current write transaction is occurring. Encoding is described in Table .
write_complete	1	In	Signal indicating that write transaction to SoPEC Unit indicated by *write_sel* is complete.

Debug Inputs from DCU			
dcu_dau_refreshcomplete	1	In	Signal indicating that the DCU has completed a refresh.
Debug Inputs from DAU IO			
various	n	In	Various DAU IO signals which can be monitored in debug mode

The CPU Interface and Arbitration Logic sub-block is shown in Figure 118.

20.14.12.1   *CPU Interface and Configuration Registers Description*

The CPU Interface and Configuration Registers sub-block provides for the CPU to access DAU specific registers by reading or writing to the DAU address space.

The CPU subsystem bus interface is described in more detail in Section 11.4.3. The DAU block will only allow supervisor mode accesses to data space (i.e. *cpu_acode[1:0]* = b11). All other accesses will result in *diu_cpu_berr* being asserted.

The configuration registers described in Section 20.14.9

Table 130. DAU configuration registers

Address (DIU_base +)	Register	#bits	Reset	Description
Reset				
0x00	Reset	1	0x1	A write to this register causes a reset of the DIU. This register can be read to indicate the reset state: 0 - reset in progress 1 - reset not in progress
Refresh				
0x04	RefreshPeriod	9	0x063	Refresh controller. When set to 0 refresh is off, otherwise the value indicates the number of cycles, less one, between each refresh. [Note that for a system clock frequency of 160MHz, a value exceeding 0x63 (indicating a 100-cycle refresh period) should not be programmed, or the DRAM will malfunction.]
Timeslot allocation and control				
0x08	NumMainTimeslots	6	0x01	Number of main timeslots (1-64) less one

0x0C	CPUPreAccessTimeslots	4	0x0	(CPUPreAccessTimeslots + 1) main slots out of a total of (CPUTotalTimeslots + 1) are preceded by a CPU access.
0x10	CPUTotalTimeslots	4	0x0	(CPUPreAccessTimeslots + 1) main slots out of a total of (CPUTotalTimeslots + 1) are preceded by a CPU access.
0x100-0x1FC	MainTimeslot[63:0]	64x4	[63:1][3:0] = 0x0 [0][3:0] = 0xE	Programmable main timeslots (up to 64 main timeslots).
0x200	ReadRoundRobinLevel	12	0x000	For each read requester plus refresh 0 = level1 of round-robin 1 = level2 of round-robin The bit order is defined in Table .
0x204	EnableCPURoundRobin	1	0x1	Allows the CPU to particpate in the unused read round-robin scheme. If disabled, the shared CPU/refresh round-robin position is dedicated solely to refresh.
0x208	RotationSync	1	0x1	Writing 0, followed by 1 to this bit allows the timeslot rotation to advance on a cycle basis which can be determined by the CPU.
0x20C	minNonCPUReadAdr	12	0x800	12 MSBs of lowest DRAM address which may be read by non-CPU requesters.
0x210	minDWUWriteAdr	12	0x800	12 MSBs of lowest DRAM address which may be written to by the DWU.
0x214	minNonCPUWriteAdr	12	0x800	12 MSBs of lowest DRAM address which may be written to by non-CPU requesters other than the DWU.
Debug				
0x300	DebugSelect[11:2]	10	0x304	Debug address select. Indicates the address of the register to report on the diu_cpu_data bus when it is not otherwise being used. When this signal carries debug

				information the signal *diu_cpu_debug_valid* will be asserted.
Debug: arbitration and performance				
0x304	ArbitrationHistory	22	-	Bit 0 = arb_gnt Bit 1 = arb_executed Bit 6:2 = arb_sel[4:0] Bit 12:7 = timeslot_number[5:0] Bit 15:13 = access_type[2:0] Bit 16 = back2back_non_cpu_write Bit 17 = sticky_back2back_non_cpu_write (Sticky version of same, cleared on reset.) Bit 18 = rotation_sync Bit 20:19 = rotation_state Bit 21 = sticky_invalid_non_cpu_adr See Section 20.14.9.2 DIU Debug for a description of the fields. Read only register.
0x308	DIUPerformance	31	-	Bit 0 = cpu_diu_rreq Bit 1 = scb_diu_rreq Bit 2 = cdu_diu_rreq Bit 3 = cfu_diu_rreq Bit 4 = lbd_diu_rreq Bit 5 = sfu_diu_rreq Bit 6 = td_diu_rreq Bit 7 = tfs_diu_rreq Bit 8 = hcu_diu_rreq Bit 9 = dnc_diu_rreq Bit 10 = llu_diu_rreq Bit 11 = pcu_diu_rreq Bit 12 = cpu_diu_wreq Bit 13 = scb_diu_wreq Bit 14 = cdu_diu_wreq Bit 15 = sfu_diu_wreq Bit 16 = dwu_diu_wreq Bit 17 = refresh_req Bit 22:18 = read_sel[4:0] Bit 23 = read_complete Bit 28:24 = write_sel[4:0]

				Bit 29 = write_complete Bit 30 = dcu_dau_refreshcomplete See Section 20.14.9.2 DIU Debug for a description of the fields. Read only register.
Debug DIU read requesters interface signals				
0x30C	CPUReadInterface	25	-	Bit 0 = cpu_diu_rreq Bit 22:1 = cpu_adr[21:0] Bit 23 = diu_cpu_rack Bit 24 = diu_cpu_rvalid Read only register.
0x310	SCBReadInterface	20		Bit 0 = scb_diu_rreq Bit 17:1 = scb_diu_radr[21:5] Bit 18 = diu_scb_rack Bit 19 = diu_scb_rvalid Read only register.
0x314	CDUReadInterface	20	-	Bit 0 = cdu_diu_rreq Bit 17:1 = cdu_diu_radr[21:5] Bit 18 = diu_cdu_rack Bit 19 = diu_cdu_rvalid Read only register.
0x318	CFUReadInterface	20	-	Bit 0 = cfu_diu_rreq Bit 17:1 = cfu_diu_radr[21:5] Bit 18 = diu_cfu_rack Bit 19 = diu_cfu_rvalid Read only register.
0x31C	LBDReadInterface	20	-	Bit 0 = lbd_diu_rreq Bit 17:1 = lbd_diu_radr[21:5] Bit 18 = diu_lbd_rack Bit 19 = diu_lbd_rvalid Read only register.
0x320	SFUReadInterface	20	-	Bit 0 = sfu_diu_rreq Bit 17:1 = sfu_diu_radr[21:5] Bit 18 = diu_sfu_rack Bit 19 = diu_sfu_rvalid Read only register.
0x324	TDReadInterface	20	-	Bit 0 = td_diu_rreq Bit 17:1 = td_diu_radr[21:5] Bit 18 = diu_td_rack

				Bit 19 = diu_td_rvalid
				Read only register.
0x328	TFSReadInterface	20	-	Bit 0 = tfs_diu_rreq
				Bit 17:1 = tfs_diu_radr[21:5]
				Bit 18 = diu_tfs_rack
				Bit 19 = diu_tfs_rvalid
				Read only register.
0x32C	HCUReadInterface	20	-	Bit 0 = hcu_diu_rreq
				Bit 17:1 = hcu_diu_radr[21:5]
				Bit 18 = diu_hcu_rack
				Bit 19 = diu_hcu_rvalid
				Read only register.
0x330	DNCReadInterface	20	-	Bit 0 = dnc_diu_rreq
				Bit 17:1 = dnc_diu_radr[21:5]
				Bit 18 = diu_dnc_rack
				Bit 19 = diu_dnc_rvalid
				Read only register.
0x334	LLUReadInterface	20	-	Bit 0 = llu_diu_rreq
				Bit 17:1 = lluu_diu_radr[21:5]
				Bit 18 = diu_llu_rack
				Bit 19 = diu_llu_rvalid
				Read only register.
0x338	PCUReadInterface	20	-	Bit 0 = pcu_diu_rreq
				Bit 17:1 = pcu_diu_radr[21:5]
				Bit 18 = diu_pcu_rack
				Bit 19 = diu_pcu_rvalid
				Read only register.
Debug DIU write requesters interface signals				
0x33C	CPUWriteInterface	27	-	Bit 0 = cpu_diu_wreq
				Bit 22:1 = cpu_adr[21:0]
				Bit 24:23 = cpu_diu_wmask[1:0]
				Bit 25 = diu_cpu_wack
				Bit 26 = cpu_diu_wvalid
				Read only register.
0x340	SCBWriteInterface	20	-	Bit 0 = scb_diu_wreq
				Bit 17:1 = scb_diu_wadr[21:5]
				Bit 18 = diu_scb_wack
				Bit 19 = scb_diu_wvalid
				Read only register.

0x344	CDUWriteInterface	22	-	Bit 0 = cdu_diu_wreq Bit 19:1 = cdu_diu_wadr[21:3] Bit 20 = diu_cdu_wack Bit 21 = cdu_diu_wvalid Read only register.
0x348	SFUWriteInterface	20	-	Bit 0 = sfu_diu_wreq Bit 17:1 = sfu_diu_wadr[21:5] Bit 18 = diu_sfu_wack Bit 19 = sfu_diu_wvalid Read only register.
0x34C	DWUWriteInterface	20	-	Bit 0 = dwu_diu_wreq Bit 17:1 = dwu_diu_wadr[21:5] Bit 18 = diu_dwu_wack Bit 19 = dwu_diu_wvalid Read only register.
Debug DAU-DCU interface signals				
0x350	DAU-DCUInterface	25	-	Bit 16:0 = dau_dcu_adr[21:5] Bit 17 = dau_dcu_rwn Bit 18 = dau_dcu_cduwpage Bit 19 = dau_dcu_refresh Bit 20 = dau_dcu_msn2stall Bit 21 = dcu_dau_adv Bit 22 = dcu_dau_wadv Bit 23 = dcu_dau_refreshcomplete Bit 24 = dcu_dau_rvalid Read only register.

are implemented here.

20.14.12.2   Arbitration Logic Description

Arbitration is triggered by the signal *re_arbitrate* from the Command Multiplexor sub-block with the signal *arb_gnt* indicating that arbitration has occurred and the arbitration winner is indicated by *arb_sel[4:0]*. The encoding of *arb_sel[4:0]* is shown in Table . The signal *dir_sel[1:0]* indicates if the arbitration winner is a read, write or refresh. Arbitration should complete within one clock cycle so *arb_gnt* is normally asserted the clock cycle after *re_arbitrate* and stays high for 1 clock cycle. *arb_sel[4:0]* and *dir_sel[1:0]* remain persistent until arbitration occurs again. The arbitration timing is shown in Figure 119.

20.14.12.2.1   Rotation Synchronisation

A configuration bit, *RotationSync*, is used to initialise advancement through the timeslot rotation, in order that the CPU will know, on a cycle basis, which timeslot is being arbitrated. This is essential for debug purposes, so that exact arbitration sequences can be reproduced.

In general, if *RotationSync* is set, slots continue to be arbitrated in the regular order specified by the timeslot rotation. When the bit is cleared, the current rotation continues until the slot pointers for pre- and main arbitration reach zero. The arbitration logic then grants DRAM access exclusively to the CPU and refreshes.

5   When the CPU again writes to *RotationSync* to cause a 0-to-1 transition of the bit, the *rdy* acknowledgment back to the CPU for this write will be exactly coincident with the RST cycle of the initial refresh which heralds the enabling of a new rotation. This refresh, along with the second access which can be either a CPU pre-access or a refresh, (depending on the CPU's request inputs), form a 2-access "preamble" before the first non-CPU requester in the new rotation can be

10   serviced. This preamble is necessary to give the write pre-arbitration the necessary head start on the main arbitration, so that write data can be loaded in time. See Figure 105 below. The same preamble procedure is followed when emerging from reset.

The alignment of *rdy* with the commencement of the rotation ensures that the CPU is always able to calculate at any point how far a rotation has progressed. *RotationSync* has a reset value of 1 to

15   ensure that the default power-up rotation can take place.

Note that any CPU writes to the DIU's other configuration registers should only be made when *RotationSync* is cleared. This ensures that accesses by non-CPU requesters to DRAM are not affected by *partial* configuration updates which have yet to be completed.

20.14.12.2.2   Motivation for Rotation Synchronisation

20   The motivation for this feature is that communications with SoPEC from external sources are synchronised to the internal clock of our position within a DIU full timeslot rotation. This means that if an external source told SOPEC to start a print 3 separate times, it would likely be at three different points within a full DIU rotation. This difference means that the DIU arbitration for each of the runs would be different, which would manifest itself externally as anomalous or inconsistent

25   print performance. The lack of reproducibility is the problem here.

However, if in response to the external source saying to start the print, we caused the internal to pass through a known state at a fixed time offset to other internal actions, this would result in reproducible prints. So, the plan is that the software would do a rotation synchronise action, then writes "Go" into various PEP units to cause the prints. This means the DIU state will be the

30   identical with respect to the PEP units state between separate runs.

20.14.12.2.3   Wind-down Protocol when Rotation Synchronisation is Initiated

When a zero is written to "RotationSync", this initiates a "wind-down protocol" in the DIU, in which any rotation already begun must be fully completed. The protocol implements the following sequence :-

35   •   The pre-arbitration logic must reach the end of whatever rotation *it* is on and stop pre-arbitrating.

    •   Only when this has happened, does the main arbitration consider doing likewise with *its* current rotation. Note that the main arbitration lags the pre-arbitration by at least 2 DRAM accesses, subject to variation by CPU pre-accesses and/or scheduled refreshes, so that the

40   two arbitration processes are sometimes on *different* rotations.

- Once the main arbitration has reached the end of its rotation, rotation synchronisation is considered to be fully activated. Arbitration then proceeds as outlined in the next section.

20.14.12.2.4   Arbitration during Rotation Synchronisation

Note that when *RotationSync* is '0' and, assuming the terminating rotation has completely drained out, then DRAM arbitration is granted according to the following fixed priority order :-

Scheduled Refresh -> CPU(W) -> CPU(R) -> Default Refresh.

CPU pre-access counters play no part in arbitration during this period. It is only subsequently, when emerging from rotation sync, that they are reloaded with the values of *CPUPreAccessTimeslots* and *CPUTotalTimeslots* and normal service resumes.

20.14.12.2.5   Timeslot-based arbitration

Timeslot-based arbitration works by having a pointer point to the current timeslot. This is shown in Figure 95 repeated here as Figure 121. When re-arbitration is signaled the arbitration winner is the current timeslot and the pointer advances to the next timeslot. Each timeslot denotes a single access. The duration of the timeslot depends on the access.

If the SoPEC Unit assigned to the current timeslot is not requesting then the unused timeslot arbitration mechanism outlined in Section 20.10.6 is used to select the arbitration winner. Note that this unused slot re-allocation is guaranteed to produce a result, because of the inclusion of refresh in the round-robin scheme.

Pseudo-code to represent arbitration is given below:

```
if re_arbitrate == 1 then
        arb_gnt = 1
        if current timeslot requesting then
                choose(arb_sel,    dir_sel)    at    current
timeslot
        else // un-used timeslot scheme
                choose   winner   according   to   un-used
timeslot allocation of Section 20.10.6
                arb_gnt = 0
```

20.14.12.3   *Arbitrating non-CPU writes in advance*

In the case of a non-CPU write commands, the write data must be transferred from the SoPEC requester before the write can occur. Arbitration should occur early to allow for any delay for the write data to be transferred to the DRAM.

Figure 113 indicates that write data transfer over 64-bit busses will take a further 4 cycles after the address is transferred. The arbitration must therefore occur 4 cycles in advance of arbitration for read accesses, Figure 109 and Figure 110, or for CPU writes Figure 112. Arbitration of CDU write accesses, Figure 114, should take place 1 cycle in advance of arbitration for read and CPU write accesses. To simplify implementation CDU write accesses are arbitrated 4 cycles in advance, similar to other non-CPU writes.

The Command Multiplexor generates a second arbitration signal *re_arbitrate_wadv* which initiates the arbitration in advance of non-CPU write accesses.

The timeslot scheme is then modified to have 2 separate pointers:

- *re_arbitrate* can arbitrate read, refresh and CPU read and write accesses according to the position of the current timeslot pointer.

- *re_arbitrate_wadv* can arbitrate only non-CPU write accesses according to the position of the write lookahead pointer.

Pseudo-code to represent arbitration is given below:

```
//re_arbitrate
if (re_arbitrate == 1) AND (current timeslot pointer!= non-
CPU write) then
    arb_gnt = 1
    if current timeslot requesting then
        choose(arb_sel, dir_sel) at current timeslot
    else // un-used read timeslot scheme
        choose  winner  according  to  un-used  read  timeslot
allocation of Section 20.10.6.2
```

If the SoPEC Unit assigned to the current timeslot is not requesting then the unused read timeslot arbitration mechanism outlined in Section 20.10.6.2 is used to select the arbitration winner.

```
//re_arbitrate_wadv
if (re_arbitrate_wadv == 1) AND (write lookahead timeslot
pointer == non-CPU write)    then
    if write lookahead timeslot requesting then
        choose(arb_sel, dir_sel) at write lookahead timeslot
        arb_gnt = 1
    elsif un-used write timeslot scheme has a requestor
        choose  winner  according  to  un-used  write  timeslot
allocation of Section 20.10.6.1
        arb_gnt = 1
    else
        //no arbitration winner
        arb_gnt = 0
```

re_arbitrate is generated in the *MSN2* state of the DCU state-machine, whereas *re_arbitrate_wadv* is generated in the *RST* state. See Figure 103.

The write lookahead pointer points two timeslots in advance of the current timeslot pointer. Therefore *re_arbitrate_wadv* causes the Arbitration Logic to perform an arbitration for non-CPU two timeslots in advance. As noted in Table , each timeslot lasts at least 3 cycles. Therefor *re_arbitrate_wadv* arbitrates at least 4 cycles in advance.

At initialisation, the write lookahead pointer points to the first timeslot. The current timeslot pointer is invalid until the write lookahead pointer advances to the third timeslot when the current timeslot pointer will point to the first timeslot. Then both pointers advance in tandem.

Some accesses can be preceded by a CPU access as in Table . These CPU accesses are not allocated timeslots. If this is the case the timeslot will last 3 (CPU access) + 3 (non-CPU access) = 6 cycles. In that case, a second write lookahead pointer, the CPU pre-access write lookahead pointer, is selected which points only one timeslot in advance. *re_arbitrate_wadv* will still arbitrate 4 cycles in advance.

20.14.12.3.1   Issuing non-CPU write commands

Although the Arbitration Logic will arbitrate non-CPU writes in advance, the Command Multiplexor must issue all accesses in the timeslot order. This is achieved as follows:

If *re_arbitrate_wadv* arbitrates a non-CPU write in advance then within the Arbitration Logic the timeslot is marked to indicate whether a write was issued.

```
//re_arbitrate_wadv
if (re_arbitrate_wadv == 1) AND (write lookahead timeslot
pointer == non-CPU write)    then
    if write lookahead timeslot requesting then
        choose(arb_sel, dir_sel) at write lookahead timeslot
        arb_gnt = 1
        MARK_timeslot = 1
    elsif un-used write timeslot scheme has a requestor
        choose  winner  according  to  un-used  write  timeslot
allocation of Section 20.10.6.1
        arb_gnt = 1
        MARK_timeslot = 1
    else
        //no pre-arbitration winner
        arb_gnt = 0
        MARK_timeslot = 0
```

When *re_arbitrate* advances to a write timeslot in the Arbitration Logic then one of two actions can occur depending on whether the slot was marked by *re_arbitrate_wadv* to indicate whether a write was issued or not.

• Non-CPU write arbitrated by *re_arbitrate_wadv*

If the timeslot has been marked as having issued a write then the arbitration logic responds to *re_arbitrate* by issuing *arb_sel[4:0], dir_sel[1:0]* and asserting *arb_gnt* as for a normal arbitration but selecting a non-CPU write access.   Normally, *re_arbitrate* does not issue non-CPU write accesses. Non-CPU writes are arbitrated by *re_arbitrate_wadv. dir_sel[1:0] == 00* indicates a non-CPU write issued by *re_arbitrate*.

• Non-CPU write not arbitrated by *re_arbitrate_wadv*

299

If the timeslot has been marked as not having issued a write, the *re_arbitrate* will use the un-used read timeslot selection to replace the un-used write timeslot with a read timeslot according to Section 20.10.6.2 Unused read timeslots allocation.

```
//re_arbitrate except for non-CPU writes
if (re_arbitrate == 1) AND (current timeslot pointer!= non-
CPU write) then
    arb_gnt = 1
    if current timeslot requesting then
        choose(arb_sel, dir_sel) at current timeslot
    else // un-used read timeslot scheme
        choose  winner  according  to  un-used  read  timeslot
allocation of Section 20.10.6.2
        arb_gnt = 1


//non-CPU write MARKED as issued
elsif (re_arbitrate == 1) AND (current timeslot pointer ==
non-CPU write) AND
        (MARK_timeslot == 1) then
        //indicate to Command Multiplexor that non-CPU write
has been arbitrated in
        //advance
        arb_gnt = 1
        dir_sel[1:0] == 00


//non-CPU write not MARKED as issued
elsif (re_arbitrate == 1) AND (current timeslot pointer ==
non-CPU write) AND
        (MARK_timeslot == 0) then
        choose  winner  according  to  un-used  read  timeslot
allocation of Section 20.10.6.2
        arb_gnt = 1
```

20.14.12.4   Flow control

If read commands are to win arbitration, the Read Multiplexor must be ready to accept the read data from the DRAM. This is indicated by the *read_cmd_rdy[1:0]* signal. *read_cmd_rdy[1:0]* supplies flow control from the Read Multiplexor.

```
        read_cmd_rdy[0]==1  //Read  multiplexor  ready  for  CPU
    read
        read_cmd_rdy[1]==1 //Read multiplexor ready for non-CPU
    read
```

The Read Multiplexor will normally always accept CPU reads, see Section 20.14.13.1, so *read_cmd_rdy[0]==1* should always apply.

Similarly, if write commands are to win arbitration, the Write Multiplexor must be ready to accept the write data from the winning SoPEC requestor. This is indicated by the *write_cmd_rdy[1:0]* signal. *write_cmd_rdy[1:0]* supplies flow control from the Write Multiplexor.

5
```
            write_cmd_rdy[0]==1  //Write  multiplexor  ready  for  CPU
        write
            write_cmd_rdy[1]==1  //Write  multiplexor  ready  for  non-
        CPU write
```

10 The Write Multiplexor will normally always accept CPU writes, see Section 20.14.13.2, so *write_cmd_rdy[0]==1* should always apply.

Non-CPU read flow control

If *re_arbitrate* selects an access then the signal *dau_dcu_msn2stall* is asserted until the Read
15 Write Multiplexor is ready.

arb_gnt is not asserted until the Read Write Multiplexor is ready.

This mechanism will stall the DCU access to the DRAM until the Read Write Multiplexor is ready to accept the next data from the DRAM in the case of a read.

20
```
        //other access flow control
        dau_dcu_msn2stall = (((re_arbitrate selects CPU read) AND
            read_cmd_rdy[0]==0) OR
                                (re_arbitrate    selects    non-CPU
        read) AND      read_cmd_rdy[1]==0))
```
25 `        arb_gnt not asserted until dau_dcu_msn2stall de-asserts`

20.14.12.5   Arbitration Hierarchy

CPU and refresh are not included in the timeslot allocations defined in the DAU configuration registers of Table .

30 The hierarchy of arbitration under normal operation is

a. CPU access

b. Refresh access

c. Timeslot access.

This is shown in Figure 124. The first DRAM access issued after reset *must* be a refresh.

35 As shown in Figure 118, the DIU request signals *<unit>_diu_rreq*, *<unit>_diu_wreq* are registered at the input of the arbitration block to ease timing. The exceptions are the *refresh_req* signal, which is generated locally in the sub-block and *cpu_diu_rreq*. The CPU read request signal is not registered so as to keep CPU DIU read access latency to a minimum. Since CPU writes are posted, *cpu_diu_wreq* is registered so that the DAU can process the write at a later juncture. The

40 arbitration logic is coded to perform arbitration of non-CPU requests first and then to gate the result with the CPU requests. In this way the CPU can make the requests available late in the arbitration cycle.

301

Note that when *RotationSync* is set to '0', a modified hierarchy of arbitration is used. This is outlined in section 20.14.12.2.3 on page 280.

20.14.12.6 Timeslot access

The basic timeslot arbitration is based on the *MainTimeslot* configuration registers. Arbitration works by the timeslot pointed to by either the current or write lookahead pointer winning arbitration. The pointers then advance to the next timeslot. This was shown in Figure 90. Each main timeslot pointer gets advanced each time it is accessed regardless of whether the slot is used.

20.14.12.7 Unused timeslot allocation

If an assigned slot is not used (because its corresponding SoPEC Unit is not requesting) then it is reassigned according to the scheme described in Section 20.10.6.

Only used non-CPU accesses are reallocated. CDU write accesses cannot be included in the unused timeslot allocation for write as CDU accesses take 6 cycles. The write accesses which the CDU write could otherwise replace require only 3 or 4 cycles.

Unused write accesses are re-allocated according to the fixed priority scheme of Table . Unused read timeslots are re-allocated according to the two-level round-robin scheme described in Section 20.10.6.2.

A pointer points to the most recently re-allocated unit in each of the round-robin levels. If the unit immediately succedling the pointer is requesting, then this unit wins the arbitration and the pointer is advanced to reflect the new winner. If this is not the case, then the subsequent units (wrapping back eventually to the pointed unit) in the level 1 round-robin are examined. When a requesting unit is found this unit wins the arbitration and the pointer is adjusted. If no unit is requesting then the pointer does not advance and the second level of round-robin is examined in a similar fashion. In the following pseudo-code the bit indices are for the *ReadRoundRobinLevel* configuration register described in Table .

```
//choose the winning arbitration level
level1 = 0
level2 = 0
for i = 0 to 11
    if unit(i) requesting AND ReadRoundRobinLevel(i) =
0 then
        level1 = 1
    if unit(i) requesting AND ReadRoundRobinLevel(i) =
1 then
        level2 = 1
```

Round-robin arbitration is effectively a priority assignment with the units assigned a priority according to the round-robin order of Table but starting at the unit currently pointed to.

```
//levelptr is pointer of selected round robin level
```

```
                    priority is array 0 to 11 // index 0 is SCBR(0) etc.
            from Table

                    //assign   decreasing   priorities   from   the   current
            pointer; maximum priority is 11
                    for i = 1 to 12
                          priority(levelptr + i) = 12 - i
                    i++
```

The arbitration winner is the one with the highest priority provided it is requesting and its
ReadRoundRobinLevel bit points to the chosen level. The *levelptr* is advanced to the arbitration
winner.

The priority comparison can be done in the hierarchical manner shown in Figure 125.

20.14.12.8   How Non-CPU Address Restrictions Affect Arbitration

Recall from Table "DAU configuration registers," on page288, " DAU configuration registers," on
page 268 that there are minimum valid DRAM addresses for non-CPU accesses, defined by
minNonCPUReadAdr, *minDWUWriteAdr* and *minNonCPUWriteAdr*. Similarly, a non-CPU
requester may not try to access a location above the high memory mark.

To ensure compliance with these address restrictions, the following DIU response occurs for any
incorrectly addressed non-CPU *writes* :-

- •   Issue a write acknowledgment at pre-arbitration time, to prevent the write requester from
       hanging.
- •   Disregard the incoming write data and write valids and void the pre-arbitration.
- •   Subsequently re-allocate the write slot at main arbitration time via the round robin.

For any incorrectly addressed non-CPU *reads*, the response is :-

- •   Arbitrate the slot in favour of the scheduled, misbehaving requester.
- •   Issue the read acknowledgement and rvalids to keep the requester from hanging.
- •   Intercept the read data coming from the DCU and send back all zeros instead.

If an invalidly addressed non-CPU access is attempted, then a sticky bit,
sticky_invalid_non_cpu_adr, is set in the *ArbitrationHistory* configuration register. See Table n
page293 on page 275 for details.

20.14.12.9   Refresh Controller Description

The refresh controller implements the functionality described in detail in Section 20.10.5. Refresh
is not included in the timeslot allocations.

CPU and refresh have priority over other accesses. If the refresh controller is requesting i.e.
refresh_req is asserted, then the refresh request will win any arbitration initiated by *re_arbitrate*.
When the refresh has won the arbitration *refresh_req* is de-asserted.

The refresh counter is reset to *RefreshPeriod[8:0]* i.e. the number of cycles between each refresh.
Every time this counter decrements to 0, a refresh is issued by asserting *refresh_req*. The counter
immediately reloads with the value in *RefreshPeriod[8:0]* and continues its countdown. It does not
wait for an acknowledgment, since the priority of a refresh request supersedes that of any

pending non-CPU access and it will be serviced immediately. In this way, a refresh *request* is guaranteed to occur every *(RefreshPeriod[8:0] + 1)* cycles. A given refresh request may incur some incidental delay in being serviced, due to alignment with DRAM accesses and the possibility of a higher-priority CPU pre-access.

5   Refresh is also included in the unused read and write timeslot allocation, having second option on awards to a round-robin position shared with the CPU. A refresh issued as a result of an unused timeslot allocation also causes the refresh counter to reload with the value in *RefreshPeriod[8:0]*. The first access issued by the DAU after reset must be a refresh. This assures that refreshes for all DRAM words fall within the required 3.2ms window.

10

```
                    //issue  a  refresh  request  if  counter  reaches  0  or  at
              reset or for re-allocated slot
                    if  RefreshPeriod  !=  0  AND  (refresh_cnt  ==  0  OR
                        diu_soft_reset_n == 0 OR
                                            prst_n            ==0              OR
              unused_timeslot_allocation == 1) then
                        refresh_req = 1
                    //de-assert refresh request when refresh acked
                    else if refresh_ack == 1 then
                        refresh_req = 0

                    //refresh counter
              if refresh_cnt == 0 OR diu_soft_reset_n == 0 OR prst_n ==0
                                            OR unused_timeslot_allocation ==
                                            1 then
                    refresh_cnt = RefreshPeriod
              else
                    refresh_cnt = refresh_cnt - 1
```

30

Refresh can preceded by a CPU access in the same way as any other access. This is controlled by the *CPUPreAccessTimeslots* and *CPUTotalTimeslots* configuration registers. Refresh will therefore not affect CPU performance. A sequence of accesses including refresh might therefore be CPU, refresh, CPU, actual timeslot.

35   *20.14.12.10   CPU Timeslot Controller Description*

CPU accesses have priority over all other accesses.CPU access is not included in the timeslot allocations. CPU access is controlled by the *CPUPreAccessTimeslots* and *CPUTotalTimeslots* configuration registers.

To avoid the CPU having to wait for its next timeslot it is desirable to have a mechanism for
40   ensuring that the CPU always gets the next available timeslot without incurring any latency on the non-CPU timeslots.

This is be done by defining each timeslot as consisting of a CPU access preceding a non-CPU access. Two counters of 4-bits each are defined allowing the CPU to get a maximum of

(CPUPreAccessTimeslots + 1) pre-accesses out of a total of (*CPUTotalTimeslots* + 1) main slots. A timeslot counter starts at *CPUTotalTimeslots* and decrements every timeslot, while another counter starts at *CPUPreAccessTimeslots* and decrements every timeslot in which the CPU uses its access. If the pre-access entitlement is used up before (*CPUTotalTimeslots* +1) slots, no

5　further CPU accesses are allowed. When the *CPUTotalTimeslots* counter reaches zero both counters are reset to their respective initial values.

When *CPUPreAccessTimeslots* is set to zero then only one pre-access will occur during every (CPUTotalTimeslots + 1) slots.

20.14.12.10.1　　Conserving CPU Pre-Accesses

10　In section 20.10.6.2.1 on page 249, it is described how the CPU can be allowed participate in the unused read round-robin scheme. When enabled by the configuration bit *EnableCPURoundRobin*, the CPU shares a joint position in the round robin with refresh. In this case, the CPU has priority, ahead of refresh, in availing of any unused slot awarded to this position.

15　Such CPU round-robin accesses do *not* count towards depleting the CPU's quota of pre-accesses, specified by *CPUPreAccessTimeslots*. Note that in order to conserve these pre-accesses, the arbitration logic, when faced with the choice of servicing a CPU request either by a pre-access or by an immediately following unused read slot which the CPU is poised to win, will opt for the latter.

20　20.14.13　Read and Write Data Multiplexor sub-block

Table 138. Read and Write Multiplexor Sub-block IO Definition

Port name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
DIU Read Interface to SoPEC Units			
diu_data	64	Out	Data from DIU to SoPEC Units except CPU. First 64-bits is bits 63:0 of 256 bit word Second 64-bits is bits 127:64 of 256 bit word Third 64-bits is bits 191:128 of 256 bit word Fourth 64-bits is bits 255:192 of 256 bit word
dram_cpu_data	256	Out	256-bit data from DRAM to CPU.
diu_<unit>_rvalid	1	Out	Signal from DIU telling SoPEC Unit that valid read data is on the *diu_data* bus
DIU Write Interface to SoPEC Units			
<unit>_diu_data	64	In	Data from SoPEC Unit to DIU except CPU. First 64-bits is bits 63:0 of 256 bit word Second 64-bits is bits 127:64 of 256 bit word

			Third 64-bits is bits 191:128 of 256 bit word
			Fourth 64-bits is bits 255:192 of 256 bit word
cpu_diu_wdatat	128	In	Write data from CPU to DIU.
<unit>_diu_wvalid	1	In	Signal from SoPEC Unit indicating that data on *<unit>_diu_data* is valid. Note that "unit" refers to non-CPU requesters only.
cpu_diu_wdatavalid	1	In	Write enable for the CPU posted write buffer. Also confirms the validity of cpu_diu_wdata.
diu_cpu_write_rdy	1	Out	Indicator that the CPU posted write buffer is empty.
Inputs from CPU Configuration and Arbitration Logic Sub-block			
arb_gnt	1	In	Signal lasting 1 cycle which indicates arbitration has occurred and *arb_sel* is valid.
arb_sel	5	In	Signal indicating which requesting SoPEC Unit has won arbitration. Encoding is described in Table .
dir_sel	2	In	Signal indicating which sense of access associated with *arb_sel* 00: issue non-CPU write 01: read winner 10: write winner 11: refresh winner
Outputs to Command Multiplexor Sub-block			
write_data_valid	2	Out	Signal indicating that valid write data is available for the current command. 00=not valid 01=CPU write data valid 10=non-CPU write data valid 11=both CPU and non-CPU write data valid
wdata	256	Out	256-bit non-CPU write data
cpu_wdata	32	Out	32-bit CPU write data
Inputs from Command Multiplexor Sub-block			
write_data_accept	2	In	Signal indicating the Command Multiplexor has accepted the write data from the write multiplexor 00=not valid 01=accepts CPU write data 10=accepts non-CPU write data 11=not valid
Inputs from DCU			
dcu_dau_rdata	256	In	256-bit read data from DCU.
dcu_dau_rvalid	1	In	Signal indicating valid read data on *dcu_dau_rdata*.

Outputs to CPU Configuration and Arbitration Logic Sub-block			
read_cmd_rdy	2	Out	Signal indicating that read multiplexor is ready for next read read command. 00=not ready 01=ready for CPU read 10=ready for non-CPU read 11=ready for both CPU and non-CPU reads
write_cmd_rdy	2	Out	Signal indicating that write multiplexor is ready for next write command. 00=not ready 01=ready for CPU write 10=ready for non-CPU write 11=ready for both CPU and non-CPU writes
Debug Outputs to CPU Configuration and Arbitration Logic Sub-block			
read_sel	5	Out	Signal indicating the SoPEC Unit for which the current read transaction is occurring. Encoding is described in Table .
read_complete	1	Out	Signal indicating that read transaction to SoPEC Unit indicated by read_sel is complete.

20.14.13.1   Read Multiplexor logic description

The Read Multiplexor has 2 read channels

- a separate read bus for the CPU, *dram_cpu_data[255:0]*.
- and a shared read bus for the rest of SoPEC, *diu_data[63:0]*.

The validity of data on the data busses is indicated by signals *diu_<unit>_rvalid*.

Timing waveforms for non-CPU and CPU DIU read accesses are shown in Figure 90 and Figure 91, respectively.

The Read Multiplexor timing is shown in Figure 127. Figure 127 shows both CPU and non-CPU reads. Both CPU and non-CPU channels are independent i.e. data can be output on the CPU read bus while non-CPU data is being transmitted in 4 cycles over the shared 64-bit read bus. CPU read data, *dram_cpu_data[255:0]*, is available in the same cycle as output from the DCU. CPU read data needs to be registered immediately on entering the CPU by a flip-flop enabled by the *diu_cpu_rvalid* signal.

To ease timing, non-CPU read data from the DCU is first registered in the Read Multiplexor by capturing it in the shared read data buffer of Figure 126 enabled by the *dcu_dau_rvalid* signal. The data is then partitioned in 64-bit words on *diu_data[63:0]*.

20.14.13.1.1   Non-CPU Read Data Coherency

Note that for data coherency reasons, a non-CPU read will always result in read data being returned to the requester which includes the after-effects of any pending (i.e. pre-arbitrated, but not yet executed) non-CPU write to the same address, which is currently cached in the non-CPU

307

write buffer. This is shown graphically in Figure n page319 on page **Error! B   kmark not defined.**.

Should the pending write be partially *masked*, then the read data returned must take account of that mask. Pending, masked writes by the CDU and SCB, as well as all unmasked non-CPU writes are fully supported.

Since *CPU* writes are dealt with on a dedicated write channel, no attempt is made to implement coherency between posted, unexecuted CPU writes and non-CPU reads to the same address.

20.14.13.1.2   Read multiplexor command queue

When the Arbitration Logic sub-block issues a read command the associated value of *arb_sel[4:0]*, which indicates which SoPEC Unit has won arbitration, is written into a buffer, the read command queue.

```
write_en = arb_gnt AND dir_sel[1:0]=="01"
if write_en==1 then
        WRITE arb_sel into read command queue
```

The encoding of *arb_sel[4:0]* is given in Table . *dir_sel[1:0]=="01"* indicates that the operation is a read. The read command queue is shown in Figure 128.

The command queue could contain values of *arb_sel[4:0]* for 3 reads at a time.

- In the scenario of Figure 127 the command queue can contain 2 values of *arb_sel[4:0]* i.e. for the simultaneous CDU and CPU accesses.

- In the scenario of Figure 130, the command queue can contain 3 values of *arb_sel[4:0]* i.e. at the time of the second *dcu_dau_rvalid* pulse the command queue will contain an *arb_sel[4:0]* for the arbitration performed in that cycle, and the two previous *arb_sel[4:0]* values associated with the data for the first two *dcu_dau_rvalid* pulses, the data associated with the first *dcu_dau_rvalid* pulse not having been fully transfered over the shared read data bus.

The read command queue is specified as 4 deep so it is never expected to fill.

The top of the command queue is a signal *read_type[4:0]* which indicates the destination of the current read data. The encoding of *read_type[4:0]* is given in Table .

20.14.13.1.3   CPU reads

Read data for the CPU goes straight out on *dram_cpu_data[255:0]* and *dcu_dau_rvalid* is output on *diu_cpu_rvalid*.

cpu_read_complete(0) is asserted when a CPU read at the top of the read command queue occurs. *cpu_read_complete(0)* causes the read command queue to be popped.

```
cpu_read_complete(0) = (read_type[4:0] == CPU read) AND
(dcu_dau_rvalid == 1)
```

If the current read command queue location points to a non-CPU access and the second read command queue location points to a CPU access then the next *dcu_dau_rvalid* pulse received is

associated with a CPU access. This is the scenario illustrated in Figure 127. The *dcu_dau_rvalid* pulse from the DCU must be output to the CPU as *diu_cpu_rvalid*. This is achieved by using *cpu_read_complete*(1) to multiplex *dcu_dau_rvalid* to *diu_cpu_rvalid*. *cpu_read_complete*(1) is also used to pop the second from top read command queue location from the read command queue.

```
cpu_read_complete(1) = (read_type == non-CPU read)
                                       AND    SECOND(read_type
== CPU read) AND (dcu_dau_rvalid == 1)
```

20.14.13.1.4   Multiplexing *dcu_dau_rvalid*

read_type[4:0] and *cpu_read_complete*(1) multiplexes the data valid signal, *dcu_dau_rvalid*, from the DCU, between the CPU and the shared read bus logic. *diu_cpu_rvalid* is the read valid signal going to the CPU. *noncpu_rvalid* is the read valid signal used by the Read Multiplexor control logic to generate read valid signals for non-CPU reads.

```
if read_type[4:0] == CPU-read then
//select CPU
   diu_cpu_rvalid:= 1
   noncpu_rvalid:= 0
if       (read_type[4:0]==       non-CPU-read)       AND
SECOND(read_type[4:0]== CPU-read)
   AND dcu_dau_rvalid == 1 then
//select CPU
   diu_cpu_rvalid:= 1
   noncpu_rvalid:= 0
else
   //select shared read bus logic
   diu_cpu_rvalid:= 0
   noncpu_rvalid:= 1
```

20.14.13.1.5   Non-CPU reads

Read data for the shared read bus is registered in the shared read data buffer using *noncpu_rvalid*. The shared read buffer has 5 locations of 64 bits with separate read pointer, *read_ptr[2:0]*, and write pointer, *write_ptr[2:0]*.

```
if noncpu_rvalid == 1 and (4 spaces in shared read
buffer) then
      shared_read_data_buffer[write_ptr]            =
dcu_dau_data[63:0]
      shared_read_data_buffer[write_ptr+1]          =
dcu_dau_data[127:64]
      shared_read_data_buffer[write_ptr+2]          =
dcu_dau_data[191:128]
```

```
                    shared_read_data_buffer[write_ptr+3]                    =
               dcu_dau_data[255:192]
```

The data written into the shared read buffer must be output to the correct SoPEC DIU read

requestor according to the value of *read_type[4:0]* at the top of the command queue. The data is

5 output 64 bits at a time on *diu_data[63:0]* according to a multiplexor controlled by *read_ptr[2:0]*.


```
               diu_data[63:0] = shared_read_data_buffer[read_ptr]
```

Figure 126 shows how *read_type[4:0]* also selects which shared read bus requesters

10 *diu_<unit>_rvalid* signal is connected to *shared_rvalid*. Since the data from the DCU is registered

in the Read Multiplexor then *shared_rvalid* is a delayed version of *noncpu_rvalid*.

When the read valid, *diu_<unit>_rvalid*, for the command associated with *read_type[4:0]* has been

asserted for 4 cycles then a signal *shared_read_complete* is asserted. This indicates that the read

has completed. *shared_read_complete* causes the value of *read_type[4:0]* in the read command

15 queue to be popped.

A state machine for shared read bus access is shown in Figure 129. This show the generation of

shared_rvalid, *shared_read_complete* and the shared read data buffer read pointer, *read_ptr[2:0]*,

being incremented.

Some points to note from Figure 129 are:

20 •   *shared_rvalid* is asserted the cycle after *dcu_dau_rvalid* associated with a shared read bus

       access. This matches the cycle delay in capturing *dau_dcu_data[255:0]* in the shared read

       data buffer. *shared_rvalid* remains asserted in the case of back to back shared read bus

       accesses.

    •   *shared_read_complete* is asserted in the last *shared_rvalid* cycle of a non-CPU access.

25   *shared_read_complete* causes the shared read data queue to be popped.

20.14.13.1.6   Read command queue read pointer logic

The read command queue read pointer logic works as follows.


```
               if shared_read_complete == 1 OR cpu_read_complete(0) == 1
30           then
                   POP top of read command queue
               if cpu_read_complete(1) == 1 then
                   POP second read command queue location
```

20.14.13.1.7   Debug signals

35 *shared_read_complete* and *cpu_read_complete* together define *read_complete* which indicates to

the debug logic that a read has completed. The source of the read is indicated on *read_sel[4:0]*.


```
               read_complete        =        shared_read_complete        OR
          cpu_read_complete(0)
40                                                OR cpu_read_complete(1)
               if cpu_read_complete(1) == 1 then
                   read_sel:= SECOND(read_type)
```

```
        else
            read_sel:= read_type
```

20.14.13.1.8   Flow control

5   There are separate indications that the Read Multiplexor is able to accept CPU and shared read
bus commands from the Arbitration Logic. These are indicated by *read_cmd_rdy[1:0]*.
The Arbitration Logic can always issue CPU reads except if the read command queue fills. The
read command queue should be large enough that this should never occur.

10
```
                    //Read Multiplexor ready for Arbitration Logic to
            issue CPU reads
                    read_cmd_rdy[0] == read command queue not full
```
For the shared read data, the Read Multiplexor deasserts the shared read bus *read_cmd_rdy[1]*
indication until a space is available in the read command queue. The read command queue
15   should be large enough that this should never occur.
read_cmd_rdy[1] is also deasserted to provide flow control back to the Arbitration Logic to keep
the shared read data bus just full.

```
                    //Read Multiplexor not ready for Arbitration Logic to
20            issue non-CPU reads
                    read_cmd_rdy[1] = (read command queue not full) AND
            (flow_control = 0)
```

The flow control condition is that DCU read data from the second of two back-to-back shared read
25   bus accesses becomes available. This causes *read_cmd_rdy[1]* to de-assert for 1 cycle, resulting
in a repeated MSN2 DCU state. The timing is shown in Figure 130.

```
        flow_control = (read_type[4:0] == non-CPU read)
                                AND SECOND(read_type[4:0] == non-
30        CPU read)

                                AND (current DCU state == MSN2)
                            AND (previous DCU state == MSN1).
```

Figure 130 shows a series of back to back transfers over the shared read data bus. The exact
35   timing of the implementation must not introduce any additional latency on shared read bus read
transfers i.e. arbitration must be re-enabled just in time to keep back to back shared read bus data
full.
The following sequence of events is illustrated in Figure 130:
* Data from the first DRAM access is written into the shared read data buffer.
40 * Data from the second access is available 3 cycles later, but its transfer into the shared read
buffer is delayed by a cycle, due to the MSN2 stall condition. (During this delay, read data

for access 2 is maintained at the output of the DRAM.) A similar 1-cycle delay is introduced for every subsequent read access until the back-to-back sequence comes to an end.

- Note that arbitration always occurs during the *last* MSN2 state of any access. So, for the second and later of any back-to-back non-CPU reads, arbitration is delayed by one cycle, i.e. it occurs every fourth cycle instead of the standard every third.

This mechanism provides flow control back to the Arbitration Logic sub-block. Using this mechanism means that the access rate will be limited to which ever takes longer - DRAM access or transfer of read data over the shared read data bus. CPU reads are always be accepted by the Read Multiplexor.

20.14.13.2   Write Multiplexor logic description

The Write Multiplexor supplies write data to the DCU.

There are two separate write channels, one for CPU data on *cpu_diu_wdata[127:0]*, one for non-CPU data on non_cpu_*wdata[255:0]*. A signal *write_data_valid[1:0]* indicates to the Command Multiplexor that the data is valid. The Command Multiplexor then asserts a signal *write_data_accept[1:0]* indicating that the data has been captured by the DRAM and the appropriate channel in the Write Multiplexor can accept the next write data.

Timing waveforms for write accesses are shown in Figure 92 to Figure 94, respectively.

There are 3 types of write accesses:

- CPU accesses

CPU write data on *cpu_diu_wdata[127:0]* is output on *cpu_wdata[127:0]*.Since CPU writes are posted, a local buffer is used to store the write data, address and mask until the CPU wins arbitration. This buffer is one position deep. *write_data_valid[0]*, which is synonymous with *!diu_cpu_write_rdy,* remains asserted until the Command Multiplexor indicates it has been written to the DRAM by asserting *write_data_accept[0]*. The CPU write buffer can then accept new posted writes.

For non-CPU writes, the Write Multiplexor multiplexes the write data from the DIU write requester to the write data buffer and the *<unit>_diu_wvalid* signal to the write multiplexor control logic.

- CDU accesses

64-bits of write data each for a masked write to a separate 256-bit word are transferred to the Write Multiplexor over 4 cycles.

When a CDU write is selected the first 64-bits of write data on *cdu_diu_wdata[63:0]* are multiplexed to non_cpu_*wdata[63:0]*. *write_data_valid[1]* is asserted to indicate a non-CPU access when *cdu_diu_wvalid* is asserted. The data is also written into the first location in the write data buffer. This is so that the data can continue to be output on non_cpu_*wdata[63:0]* and *write_data_valid[1]* remains asserted until the Command Multiplexor indicates it has been written to the DRAM by asserting *write_data_accept[1]*. Data continues to be accepted from the CDU and is written into the other locations in the write data buffer. Successive *write_data_accept[1]* pulses cause the successive 64-bit data words to be output on *wdata[63:0]* together with *write_data_valid[1]*. The last *write_data_accept[1]* means the write buffer is empty and new write data can be accepted.

312

- Other write accesses.

256-bits of write data are transferred to the Write Multiplexor over 4 successive cycles. When a write is selected the first 64-bits of write data on *<unit>_diu_wdata[63:0]* are written into the write data buffer. The next 64-bits of data are written to the buffer in successive cycles. Once the last 64-bit word is available on *<unit>_diu_wdata[63:0]* the entire word is output on non_cpu_*wdata[255:0]*, write_*data_valid [1]* is asserted to indicate a non-CPU access, and the last 64-bit word is written into the last location in the write data buffer. Data continues to be output on non_cpu_*wdata[255:0]* and *write_data_valid[1]* remains asserted until the Command Multiplexor indicates it has been written to the DRAM by asserting *write_data_accept[1]*. New write data can then be written into the write buffer.

CPU write multiplexor control logic

When the Command Multiplexor has issued the CPU write it asserts *write_data_accept[0]*. *write_data_accept[0]* causes the write multiplexor to assert *write_cmd_rdy[0]*.

The signal *write_cmd_rdy[0]* tells the Arbitration Logic sub-block that it can issue another CPU write command i.e. the CPU write data buffer is empty.

Non-CPU write multiplexor control logic

The signal *write_cmd_rdy[1]* tells the Arbitration Logic sub-block that the Write Multiplexor is ready to accept another non-CPU write command. When *write_cmd_rdy[1]* is asserted the Arbitration Logic can issue a write command to the Write Multiplexor. It does this by writing the value of *arb_sel[4:0]* which indicates which SoPEC Unit has won arbitration into a write command register, *write_cmd[3:0]*.

```
        write_en = arb_gnt AND dir_sel[1]==1 AND arb_sel = non-
        CPU
        if write_en==1 then
          write_cmd = arb_sel
```

The encoding of *arb_sel[4:0]* is given in Table . *dir_sel[1]==1* indicates that the operation is a write. *arb_sel[4:0]* is only written to the write command register if the write is a non-CPU write.

A rule was introduced in Section 20.7.2.3 Interleaving read and write accesses to the effect that non-CPU write accesses would not be allocated adjacent timeslots. This means that a single write command register is required.

The write command register, *write_cmd[3:0]*, indicates the source of the write data. *write_cmd[3:0]* multiplexes the write data *<unit>_diu_wdata*, and the data valid signal, *<unit>_diu_wvalid*, from the selected write requestor to the write data buffer. Note, that CPU write data is not included in the multiplex as the CPU has its own write channel. The *<unit>_diu_wvalid* are counted to generate the signal *word_sel[1:0]* which decides which 64-bit word of the write data buffer to store the data from *<unit>_diu_wdata*.

```
        //when the Command Multiplexor accepts the write data
        if write_data_accept[1] = 1 then
```

313

```
            //reset the word select signal
              word_sel[1:0]=00
        //when wvalid is asserted
        if wvalid = 1 then
            //increment the word select signal
            if word_sel[1:0] == 11 then
              word_sel[1:0] == 00
          else
              word_sel[1:0] == word_sel[1:0] + 1
```

wvalid is the *<unit>_diu_wvalid* signal multiplexed by *write_cmd[3:0]*. *word_sel[1:0]* is reset when the Command Multiplexor accepts the write data. This is to ensure that *word_sel[1:0]* is always starts at 00 for the first *wvalid* pulse of a 4 cycle write data transfer.

The write command register is able to accept the next write when the Command Multiplexor accepts the write data by asserting *write_data_accept[1]*. Only the last *write_data_accept[1]* pulse associated with a CDU access (there are 4) will cause the write command register to be ready to accept the next write data.

Flow control back to the Command Multiplexor

write_cmd_rdy[0] is asserted when the CPU data buffer is empty.

write_cmd_rdy[1] is asserted when both the write command register and the write data buffer is empty.

PEP SUBSYSTEM

21   PEP Controller Unit (PCU)

21.1   OVERVIEW

The PCU has three functions:

- The first is to act as a bus bridge between the CPU-bus and the PCU-bus for reading and writing PEP configuration registers.

- The second is to support page banding by allowing the PEP blocks to be reprogrammed between bands by retrieving commands from DRAM instead of being programmed directly by the CPU.

- The third is to send register debug information to the RDU, within the CPU subsystem, when the PCU is in Debug Mode.

21.2   INTERFACES BETWEEN PCU AND OTHER UNITS

21.3   BUS BRIDGE

The PCU is a bus-bridge between the CPU-bus and the PCU-bus. The PCU is a slave on the CPU-bus but is the only master on the PCU-bus. See Figure   page39 on page **Error! Bookmark not defined.**.

21.3.1   CPU accessing PEP

All the blocks in the PEP can be addressed by the CPU via the PCU. The MMU in the CPU-subsystem will decode a PCU select signal, *cpu_pcu_sel*, for all the PCU mapped addresses (see section 11.4.3 on page 69). Using *cpu_adr* bits 15-12 the PCU will decode individual block selects for each of the blocks within the PEP. The PEP blocks then decode the remaining address bits

needed to address their PCU-bus mapped registers. Note: the CPU is only permitted to perform supervisor-mode data-type accesses of the PEP, i.e. *cpu_acode* = 11. If the PCU is selected by the CPU and any other code is present on the *cpu_acode* bus the access is ignored by the PCU and the *pcu_cpu_berr* signal is strobed,

5   CPU commands have priority over DRAM commands. When the PCU is executing each set of four commands retrieved from DRAM the CPU can access PCU-bus registers. In the case that DRAM commands are being executed and the CPU resets the *CmdSource* to zero, the contents of the DRAM *CmdFifo* is invalidated and no further commands from the fifo are executed. The *CmdPending* and *NextBandCmdEnable* work registers are also cleared.

10   When a DRAM command writes to the *CmdAdr* register it means the next DRAM access will occur at the address written to *CmdAdr*. Therefore if the JUMP instruction is the first command in a group of four, the other three commands get executed and then the PCU will issue a read request to DRAM at the address specified by the JUMP instruction. If the JUMP instruction is the second command then the following two commands will be executed before the PCU requests

15   from the new DRAM address specified by the JUMP instruction etc.Therefore the PCU will always execute the remaining commands in each four command group before carrying out the JUMP instruction.

21.4   PAGE BANDING

The PCU can be programmed to associate microcode in DRAM with each *finishedband* signal.

20   When a *finishedband* signal is asserted the PCU will read commands from DRAM and execute these commands. These commands are each 64-bits (see Section 21.8.5) and consist of 32-bit address bits and 32 data bits and allow PCU mapped registers to be programmed directly by the PCU.

If more than one *finishedband* signal is received at the same time, or others are received while

25   microcode is already executing, the PCU will hold the commands as pending, and will execute them at the first opportunity.

Each microcode program associated with *cdu_finishedband*, *lbd_finishedband* and *te_finishedband* would simply restart the appropriate unit with new addresses - a total of about 4 or 5 microcode instructions. As well, or alternatively, *pcu_finishedband* can be used to set up all of

30   the units and therefore involves many more instructions. This minimizes the time that a unit is idle in between bands. The *pcu_finishedband* control signal is issued once the specified combination of CDU, LBD and TE (programmed in *BandSelectMask*) have finished their processing for a band.

21.5   INTERRUPTS, ADDRESS LEGALITY AND SECURITY

Interrupts are generated when the various page expansion units have finished a particular band of

35   data from DRAM. The *cdu_finishedband, lbd_finishedband* and *te_finishedband* signals are combined in the PCU into a single interrupt *pcu_finishedband* which is exported by the PCU to the interrupt controller.

The PCU mapped registers should only be accessible from Supervisor Data Mode. The area of DRAM where PCU commands are stored should be a Supervisor Mode only DRAM area,

40   although this is not enforced by the PCU.

When the PCU is executing commands from DRAM, any block-address decoded from a command which is not part of the PEP block-address map will cause the PCU to ignore the command and strobe the *pcu_icu_address_invalid* interrupt signal. The CPU can then interrogate the PCU to find the source of the illegal command. The MMU will ensure that the CPU cannot

5   address an invalid PEP subsystem block.

When the PCU is executing commands from DRAM, any address decoded from a command which is not part of the PEP address map will cause the PCU to:

- Cease execution of current command and flush all remaining commands already retrieved from DRAM.

10   - Clear *CmdPending* work-register.

- Clear *NextBandCmdEnable* registers.

- Set *CmdSource* to zero.

In addition to cancelling all current and pending DRAM accesses the PCU strobes the *pcu_icu_address_invalid* interrupt signal. The CPU can then interrogate the PCU to find the

15   source of the illegal command.

21.6   DEBUG MODE

When the need to monitor the (possibly changing) value in any PEP configuration register the PCU may be placed in Debug Mode. This is done via the CPU setting certain Debug Address register within the PCU. Once in Debug Mode the PCU continually reads the target PEP

20   configuration register and sends the read value to the RDU. Debug Mode has the lowest priority of all PCU functions: if the CPU wishes to perform an access or there are DRAM commands to be executed they will interrupt the Debug access, and the PCU will resume Debug access once a CPU or DRAM command has completed.

21.7   IMPLEMENTATION

25   21.7.1   Definitions of I/O

Table 139. PCU Port List

Port Name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	SoPEC functional clock
prst_n	1	In	Active-low, synchronous reset in pclk domain
End of Band Functionality			
cdu_finishedband	1	In	Finished band signal from CDU
lbd_finishedband	1	In	Finished band signal from LBD
te_finishedband	1	In	Finished band signal from TE
pcu_finishedband	1	Out	Asserted once the specified combination of CDU, LBD, and TE have finished their processing for a band.
PCU address error			

pcu_icu_address_invalid	1	Out	Strobed if PCU decodes a non PEP address from commands retrieved from DRAM or CPU.
CPU Subsystem Interface Signals			
cpu_adr[15:2]	14	In	CPU address bus. 14 bits are required to decode the address space for the PEP.
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
pcu_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
cpu_pcu_sel	1	In	Block select from the CPU. When *cpu_pcu_sel* is high both *cpu_adr* and *cpu_dataout* are valid
pcu_cpu_rdy	1	Out	Ready signal to the CPU. When *pcu_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the block and for a read cycle this means the data on *pcu_cpu_data* is valid.
pcu_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
pcu_cpu_debug_valid	1	Out	Debug Data valid on *pcu_cpu_data* bus. Active high.
PCU Interface to PEP blocks			
pcu_adr[11:2]	10	Out	PCU address bus. The 10 least significant bits of *cpu_adr [15:2]* allow 1024 32-bit word addressable locations per PEP block. Only the number of bits required to decode the address space are exported to each block.
pcu_dataout[31:0]	32	Out	Shared write data bus from the PCU
<unit>_pcu_datain[31:0]	32	In	Read data bus from each PEP subblock to the PCU
pcu_rwn	1	Out	Common read/not-write signal from the PCU
pcu_<unit>_sel	1	Out	Block select for each PEP block from the PCU. Decoded from the 4 most significant bits of *cpu_adr[15:2]*. When *pcu_<unit>_sel* is high both *pcu_adr* and *pcu_dataout* are valid
<unit>_pcu_rdy	1	In	Ready from each PEP block signal to the PCU. When *<unit>_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means

			pcu_dataout has been registered by the block and for a read cycle this means the data on <unit>_pcu_datain is valid.
DIU Read Interface signals			
pcu_diu_rreq	1	Out	PCU requests DRAM read. A read request must be accompanied by a valid read address.
pcu_diu_radr[21:5]	17	Out	Read address to DIU 17 bits wide (256-bit aligned word).
diu_pcu_rack	1	In	Acknowledge from DIU that read request has been accepted and new read address can be placed on pcu_diu_radr
diu_data[63:0]	64	In	Data from DIU to PCU. First 64-bits is bits 63:0 of 256 bit word Second 64-bits is bits 127:64 of 256 bit word Third 64-bits is bits 191:128 of 256 bit word Fourth 64-bits is bits 255:192 of 256 bit word
diu_pcu_rvalid	1	In	Signal from DIU telling PCU that valid read data is on the diu_data bus

21.7.2   Configuration Registers

Table 140. PCU Configuration Registers

Address PCU_base+	register	#bits	reset	description
Control registers				
0x00	Reset	1	0x1	A write to this register causes a reset of the PCU. This register can be read to indicate the reset state: 0 - reset in progress 1 - reset not in progress
0x04	CmdAdr[21:5] (256-bit aligned DRAM address)	17	0x00 000	The address of the next set of commands to retrieve from DRAM. When this register is written to, either by the CPU or DRAM command, 1 is also written to CmdSource to cause the execution of the commands at the specified address.
0x08	BandSelect Mask[2:0]	3	0x0	Selects which input finishedBand flags are to be watched to generate the combined pcu_finishedband signal.

318

				BitO - lbd_finishedband
				Bit1 - cdu_finishedband
				Bit2 - te_finishedband
0x0C, 0x10, 0x14, 0x18	NextBandCmdAdr[3:0][21:5] (256-bit aligned DRAM address)	4x17	0x00 000	The address to transfer to *CmdAdr* as soon as possible after the next *finishedBand[n]* signal has been received as long as *NextBandCmdEnable[n]* is set. A write from the PCU to *NextBandCmdAdr[n]* with a non-zero value also sets *NextBandCmdEnable[n]*. A write from the PCU to *NextBandCmdAdr[n]* with a 0 value clears *NextBandCmdEnable[n]*.
0x1C	NextCmdAdr[21:5]	17	0x00 000	The address to transfer to *CmdAdr* when the CPU pending bit (*CmdPending[4]*) get serviced. A write from the PCU to *NextCmdAdr[n]* with a non-zero value also sets *CmdPending[4]*. A write from the PCU to *NextCmdAdr[n]* with a 0 value clears *CmdPending[4]*
0x20	CmdSource	1	0x0	0 - commands are taken from the CPU 1 - commands are taken from the CPU as well as DRAM at *CmdAdr*.
0x24	DebugSelect[15:2]	14	0x00 00	Debug address select. Indicates the address of the register to report on the *pcu_cpu_data* bus when it is not otherwise being used, and the PEP bus is not being used Bits [15:12] select the unit (see Table) Bits [11:2] select the register within the unit
Work registers (read only)				
0x28	InvalidAddress[21:3] (64-bit aligned DRAM)	19	0	DRAM Address of current 64-bit command attempting to execute. Read only register.
0x2C	CmdPending	5	0x00	For each bit n, where n is 0 to 3 0 -no commands pending for *NextBandCmdAdr[n]* 1 -commands pending for *NextBandCmdAdr[n]*

319

				For bit 4 0 -no commands pending for *NextCmdAdr[n]* 1 -commands pending for *NextCmdAdr[n]* Read only register.
0x34	FinishedSo Far	3	0x0	The appropriate bit is set whenever the corresponding input finishedBand flag is set and the corresponding bit in the *BandSelectMask* bit is also set. If all *FinishedSoFar* bits are set wherever *BandSelect* bits are also set, all *FinishedSoFar* bits are cleared and the output *pcu_finishedband* signal is given. Read only register.
0x38	NextBandC mdEnable	4	0x0	This register can be written to indirectly (i.e. the bits are set or cleared via writes to *NextBandCmdAdr[n]*) For each bit: 0 - do nothing at the next *finishedBand[n]* signal. 1 - Execute instructions at *NextBandCmdAdr[n]* as soon as possible after receipt of the next *finishedBand[n]* signal. Bit0 - lbd_finishedband Bit1 - cdu_finishedband Bit2 - te_finishedband Bit3 - pcu_finishedband Read only register.

21.8   DETAILED DESCRIPTION

21.8.1   PEP Blocks Register Map

All PEP accesses are 32-bit register accesses.

From Table 140 it can be seen that four bits only are necessary to address each of the sub-blocks within the PEP part of SoPEC. Up to 14 bits may be used to address any configurable 32-bit register within PEP. This gives scope for 1024 configurable registers per sub-block. This address will come either from the CPU or from a command stored in DRAM. The bus is assembled as follows:

- adr[15:12] = sub-block address
- adr[n:2] = 32-bit register address within sub-block, only the number of bits required to decode the registers within each sub-block are used.

Table 141. PEP blocks Register Map

Block	Block Select Decode = cpu_adr[15:12]
PCU	0x0
CDU	0x1
CFU	0x2
LBD	0x3
SFU	0x4
TE	0x5
TFU	0x6
HCU	0x7
DNC	0x8
DWU	0x9
LLU	0xA
PHI	0xB
Reserved	0xC to 0xF

21.8.2   Internal PCU PEP protocol

The PCU performs PEP configuration register accesses via a select signal, *pcu_<block>_sel*. The read/ write sense of the access is communicated via the *pcu_rwn* signal (1 = read, 0 = write).

5   Write data is clocked out, and read data clocked in upon receipt of the appropriate select-read/write-address combination.

Figure 133 shows a write operation followed by a read operation. The read operation is shown with wait states while the PEP block returns the read data.

For access to the PEP blocks a simple bus protocol is used. The PCU first determines which

10   particular PEP block is being addressed so that the appropriate block select signal can be generated. During a write access PCU write data is driven out with the address and block select signals in the first cycle of an access. The addressed PEP block responds by asserting its ready signal indicating that it has registered the write data and the access can complete. The write data bus is common to all PEP blocks.

15   A read access is initiated by driving the address and select signals during the first cycle of an access. The addressed PEP block responds by placing the read data on its bus and asserting its ready signal to indicate to the PCU that the read data is valid. Each block has a separate point-to-point data bus for read accesses to avoid the need for a tri-stateable bus.

Consecutive accesses to a PEP block must be separated by at least a single cycle, during which

20   the select signal must be de-asserted.

21.8.3   PCU DRAM access requirements

The PCU can execute register programming commands stored in DRAM. These commands can be executed at the start of a print run to initialize all the registers of PEP. The PCU can also execute instructions at the start of a page, and between bands. In the inter-band time, it is critical

to have the PCU operate as fast as possible. Therefore in the inter-page and inter-band time the PCU needs to get low latency access to DRAM.

A typical band change requires on the order of 4 commands to restart each of the CDU, LBD, and TE, followed by a single command to terminate the DRAM command stream. This is on the order of 5 commands per restart component.

The PCU does single 256 bit reads from DRAM. Each PCU command is 64 bits so each 256 bit DRAM read can contain 4 PCU commands. The requested command is read from DRAM together with the next 3 contiguous 64-bits which are cached to avoid unnecessary DRAM reads. Writing zero to *CmdSource* causes the PCU to flush commands and terminate program access from DRAM for that command stream. The PCU requires a 256-bit buffer to the 4 PCU commands read by each 256-bit DRAM access. When the buffer is empty the PCU can request DRAM access again. Adding a 256-bit double buffer would allow the next set of 4 commands to be fetched from DRAM while the current commands are being executed.

1024 commands of 64 bits requires 8 kB of DRAM storage.

Programs stored in DRAM are referred to as *PCU Program Code*.

21.8.4   End of band unit

The state machine is responsible for watching the various input *xx_finishedband* signals, setting the *FinishedSoFar* flags, and outputting the *pcu_finishedband* flags as specified by the *BandSelect* register.

Each cycle, the end of band unit performs the following tasks:

```
pcu_finishedband   = (FinishedSoFar[0]  == BandSelectMask[0])
AND
                     (FinishedSoFar[1]                    ==
BandSelectMask[1]) AND
                     (FinishedSoFar[2]                    ==
BandSelectMask[2]) AND
                     (BandSelectMask[0]                   OR
BandSelectMask[1] OR BandSelectMask[2])
     if (pcu_finishedband == 1) then
       FinishedSoFar[0] = 0
       FinishedSoFar[1] = 0
       FinishedSoFar[2] = 0
     else
       FinishedSoFar[0]       =       (FinishedSoFar[0]   OR
lbd_finishedband) AND BandSelectMask[0]
       FinishedSoFar[1]       =       (FinishedSoFar[1]   OR
cdu_finishedband) AND BandSelectMask[1]
       FinishedSoFar[2]       =       (FinishedSoFar[2]   OR
te_finishedband) AND BandSelectMask[2]
```

Note that it is the responsibility of the microcode at the start of printing a page to ensure that all 3 *FinishedSoFar* bits are cleared. It is not necessary to clear them between bands since this happens automatically.

If a bit of *BandSelectMask* is cleared, then the corresponding bit of *FinishedSoFar* has no impact on the generation of *pcu_finishedband*.

21.8.5   Executing commands from DRAM

Registers in PEP can be programmed by means of simple 64-bit commands fetched from DRAM. The format of the commands is given in Table 142. Register locations can have a data value of up to 32 bits. Commands are PEP register write commands only.

Table 142. Register write commands in PEP

command	bits 63-32	bits 31-16	bits 15-2	bits 1-0
Register write	data	zero	32-bit word address	zero

Due attention must be paid to the endianness of the processor. The LEON processor is a big-endian processor (bit 7 is the most significant bit).

21.8.6   General Operation

Upon a Reset condition, *CmdSource* is cleared (to 0), which means that all commands are initially sourced only from the CPU bus interface. Registers and can then be written to or read from one location at a time via the CPU bus interface.

If *CmdSource* is 1, commands are sourced from the DRAM at *CmdAdr* and from the CPU bus. Writing an address to *CmdAdr* automatically sets *CmdSource* to 1, and causes a command stream to be retrieved from DRAM. The PCU will execute commands from the CPU or from the DRAM command stream, giving higher priority to the CPU always.

If *CmdSource* is 0 the DRAM requestor examines the *CmdPending* bits to determine if a new DRAM command stream is pending. If any of *CmdPending* bits are set, then the appropriate *NextBandCmdAdr* or *NextCmdAdr* is copied to *CmdAdr* (causing *CmdSource* to get set to 1) and a new command DRAM stream is retrieved from DRAM and executed by the PCU. If there are multiple pending commands the DRAM requestor will service the lowest number pending bit first. Note that a new DRAM command stream only gets retrieved when the current command stream is empty.

If there are no DRAM commands pending, and no CPU commands the PCU defaults to an idle state. When idle the PCU address bus defaults to the *DebugSelect* register value (bits 11 to 2 in particular) and the default unit PCU data bus is reflected to the CPU data bus. The default unit is determined by the *DebugSelect* register bits 15 to 12.

In conjunction with this, upon receipt of a *finishedBand[n]* signal, *NextBandCmdEnable[n]* is copied to *CmdPending[n]* and *NextBandCmdEnable[n]* is cleared. Note, each of the LBD, CDU, and TE (where present) may be re-programmed individually between bands by appropriately setting *NextBandCmdAdr[2-0]* respectively. However, execution of inter-band commands may be postponed until all blocks specified in the *BandSelectMask* register have pulsed their *finishedband*

signal. This may be accomplished by only setting *NextBandCmdAdr[3]* (indirectly causing *NextBandCmdEnable[3]* to be set) in which case it is the *pcu_finishedband* signal which causes *NextBandCmdEnable[3]* to be copied to *CmdPending[3]*.

To conveniently update multiple registers, for example at the start of printing a page, a series of Write Register commands can be stored in DRAM. When the start address of the first Write Register command is written to the *CmdAdr* register (via the CPU), the *CmdSource* register is automatically set to 1 to actually start the execution at *CmdAdr*. Alternatively the CPU can write to *NextCmdAdr* causing the *CmdPending[4]* bit to get set, which will then get serviced by the DRAM requestor in the pending bit arbitration order.

The final instruction in the command block stored in DRAM must be a register write of 0 to *CmdSource* so that no more commands are read from DRAM. Subsequent commands will come from pending programs or can be sent via the CPU bus interface.

21.8.6.1 Debug Mode

Debug mode is implemented by reusing the normal CPU and DRAM access decode logic. When in the *Arbitrate* state (see state machine A below), the PEP address bus is defaulted to the value in the *DebugSelect* register. The top bits of the *DebugSelect* register are used to decode a select to a PEP unit and the remaining bits are reflected on the PEP address bus. The selected units read data bus is reflected on the *pcu_cpu_data* bus to the RDU in the CPU. The *pcu_cpu_debug_valid* signal indicates to the RDU that the data on the *pcu_cpu_data* bus is valid debug data.

Normal CPU and DRAM command access will require the PEP bus, and as such will cause the debug data to be invalid during the access, this is indicated to the RDU by setting *pcu_cpu_debug_valid* to zero.

The decode logic is :

```
// Default Debug decode
if state == Arbitrate then
    if    (cpu_pcu_sel    ==    1    AND    cpu_acode    /=
SUPERVISOR_DATA_MODE) then
        pcu_cpu_debug_valid        = 0            // bus  error
condition
        pcu_cpu_data            = 0
    else
        <unit>                = decode(DebugSelect[15:12])
        if (<unit> == PCU ) then
            pcu_cpu_data        = Internal PCU register
        else
            pcu_cpu_data        = <unit>_pcu_datain[31:0]
        pcu_adr[11:2]        = DebugSelect[11:2]
        pcu_cpu_debug_valid        = 1 AFTER 4 clock cycles
    else
        pcu_cpu_debug_valid        = 0
```

21.8.7   State Machines

DRAM command fetching and general command execution is accomplished using two state machines. State machine A evaluates whether a CPU or DRAM command is being executed, and proceeds to execute the command(s). Since the CPU has priority over the DRAM it is permitted to interrupt the execution of a stream of DRAM commands.

5   Machine B decides which address should be used for DRAM access, fetches commands from DRAM and fills a command fifo which A executes. The reason for separating the two functions is to facilitate the execution of CPU or Debug commands while state machine B is performing DRAM reads and filling the command fifo. In the case where state machine A is ready to execute commands (in its *Arbitrate* state) and it sees both a full DRAM command fifo and an active

10   *cpu_pcu_sel* then the DRAM commands are executed last.

21.8.7.1 State Machine A: Arbitration and execution of commands

The state-machine enters the *Reset* state when there is an active strobe on either the reset pin, *prst_n*, or the PCU's soft-reset register. All registers in the PCU are zeroed, unless otherwise specified, on the next rising clock edge. The PCU self-deasserts the soft reset in the *pclk* cycle

15   after it has been asserted.

The state changes from *Reset* to *Arbitrate* when *prst_n* == 1 and *PCU_softreset* == 1.

The state-machine waits in the *Arbitrate* state until it detects a request for CPU access to the PEP units (*cpu_pcu_sel* == 1 and *cpu_acode* == 11) or a request to execute DRAM commands *CmdSource* == 1, and DRAM commands are available, *CmdFifoFull*==1. Note if (*cpu_pcu_sel* ==

20   1 and *cpu_acode* != 11) the CPU is attempting an illegal access. The PCU ignores this command and strobes the *cpu_pcu_berr* for one cycle.

While in the *Arbitrate* state the machine assigns the *DebugSelect* register to the PCU unit decode logic and the remaining bits to the PEP address bus. When in this state the debug data returned from the selected PEP unit is reflected on the CPU bus (*pcu_cpu_data* bus) and the

25   *pcu_cpu_debug_valid*=1.

If a CPU access request is detected (*cpu_pcu_sel* == 1 and *cpu_acode* == 11) then the machine proceeds to the *CpuAccess* state. In the *CpuAccess* state the cpu address is decoded and used to determine the PEP unit to select. The remaining address bits are passed through to the PEP address bus. The machine remains in the *CpuAccess* state until a valid ready from the selected

30   PEP unit is received. When received the machine returns to the arbitrate state, and the ready signal to the CPU is pulsed.

```
// decode the logic
pcu_<unit>_sel = decode(cpu_adr[15:12])
pcu_adr[11:2]  = cpu_adr[11:2]
```

35   The CPU is prevented from generating an invalid PEP unit address (prevented in the MMU) and so CPU accesses cannot generate an invalid address error.

If the state machine detects a request to execut DRAM commands (*CmdSource* == 1), it will wait in the *Arbitrate* state until commands have been loaded into the command FIFO from DRAM (all controlled by state machine B). When the DRAM commands are available (*cmd_fifo_full* == 1) the

40   state machine will proceed to the *DRAMAccess* state.

When in the *DRAMAccess* state the commands are executed from the *cmd_fifo*. A command in the *cmd_fifo* consists of 64-bits (or which the FIFO holds 4). The decoding of the 64-bits to commands is given in Table . For each command the decode is

```
// DRAM command decode
pcu_<unit>_sel = decode( cmd_fifo[cmd_count][15:12] )
pcu_adr[11:2]  = cmd_fifo[cmd_count][11:2]
pcu_dataout    = cmd_fifo[cmd_count][63:32]
```

When the selected PEP unit returns a ready signal (*<unit>_pcu_rdy*==1) indicating the command has completed, the state machine will return to the *Arbitrate* state. If more commands exists (*cmd_count* !=0) the transition will decrement the command count.

When in the *DRAMAccess* state, if when decoding the DRAM command address bus (*cmd_fifo[cmd_count][15:12]*), the address selects a reserved address, the state machine proceeds to the *AdrError* state, and then back to the *Arbitrate* state. An address error interrupt will be generated and the DRAM command FIFOs will be cleared.

A CPU access can pre-empt any pending DRAM commands. After each command is completed the state machine returns to the *Arbitrate* state. If a CPU access is required and DRAM command stream is executing the CPU access always takes priority. If a CPU or DRAM command sets the *CmdSource* to 0, all subsequent DRAM commands in the command FIFO are cleared. If the CPU sets the *CmdSource* to 0 the *CmdPending* and *NextBandCmdEnable* work registers are also cleared.

21.8.7.2 State Machine B: Fetching DRAM commands

A system reset (*prst_n*==0) or a software reset (*pcu_softreset_n*==0) will cause the state machine to reset to the *Reset* state. The state machine remains in the *Reset* until both reset conditions are removed. When removed the machine proceeds to the *Wait* state.

The state machine waits in the *Wait* state until it determines that commands are needed from DRAM. Two possible conditions exist that require DRAM access. Either the PCU is processing commands which must be fetched from DRAM (*cmd_source*==1), and the command FIFO is empty (*cmd_fifo_full*==0), or the *cmd_source*==0 and the command FIFO is empty and there are some commands pending (*cmd_pending* !=0). In either of these conditions the machine proceeds to the *Ack* state and issues a read request to DRAM (*pcu_diu_rreq*==1), it calculates the address to read from dependent on the transition condition. In the command pending transition condition, the highest priority *NextBandCmdAdr* (or *NextCmdAdr*) that is pending is used for the read address (*pcu_diu_radr*) and is also copied to the *CmdAdr* register. If multiple pending bits are set the lowest pending bits are serviced first. In the normal PCU processing transition the *pcu_diu_radr* is the *CmdAdr* register.

When an acknowledge is received from the DRAM the state machine goes to the *FillFifo* state. In the *FillFifo* state the machine waits for the DRAM to respond to the read request and transfer data words. On receipt of the first word of data *diu_pcu_rvalid*==1, the machine stores the 64-bit data word in the command FIFO (*cmd_fifo[3]*) and transitions to the *Data1*, *Data2*, *Data3* states each time waiting for a *diu_pcu_rvalid*==1 and storing the transferred data word to *cmd_fifo[2]*, *cmd_fifo[1]* and *cmd_fifo[0]* respectively.

When the transfer is complete the machine returns to the *Wait* state, setting the *cmd_count* to 3, the *cmd_fifo_full* is set to 1 and the *CmdAdr* is incremented.

If the CPU sets the *CmdSource* register low while the PCU is in the middle of a DRAM access, the statemachine returns to the *Wait* state and the DRAM access is aborted.

5   *21.8.7.3 PCU_ICU_Address_Invalid Interrupt*

When the PCU is executing commands from DRAM, addresses decoded from commands which are not PCU mapped addresses (4-bits only) will result in the current command being ignored and the *pcu_icu_address_invalid* interrupt signal is strobed. When an invalid command occurs all remaining commands already retrieved from DRAM are flushed from the *CmdFifo,* and the

10   *CmdPending, NextBandCmdEnable* and *CmdSource* registers are cleared to zero.

The CPU can then interrogate the PCU to find the source of the illegal DRAM command via the *InvalidAddress* register.

The CPU is prevented by the MMU from generating an invalid address command.

22   Contone Decoder Unit (CDU)

15   22.1   OVERVIEW

The Contone Decoder Unit (CDU) is responsible for performing the optional decompression of the contone data layer.

The input to the CDU is up to 4 planes of compressed contone data in JPEG interleaved format. This will typically be 3 planes, representing a CMY contone image, or 4 planes representing a

20   CMYK contone image. The CDU must support a page of A4 length (11.7 inches) and Letter width (8.5 inches) at a resolution of 267 ppi in 4 colors and a print speed of 1 side per 2 seconds.

The CDU and the other page expansion units support the notion of page banding. A compressed page is divided into one or more bands, with a number of bands stored in memory. As a band of the page is consumed for printing a new band can be downloaded. The new band may be for the

25   current page or the next page. Band-finish interrupts have been provided to notify the CPU of free buffer space.

The compressed contone data is read from the on-chip DRAM. The output of the CDU is the decompressed contone data, separated into planes. The decompressed contone image is written to a circular buffer in DRAM with an expected minimum size of 12 lines and a configurable

30   maximum. The decompressed contone image is subsequently read a line at a time by the CFU, optionally color converted, scaled up to 1600 ppi and then passed on to the HCU for the next stage in the printing pipeline. The CDU also outputs a *cdu_finishedband* control flag indicating that the CDU has finished reading a band of compressed contone data in DRAM and that area of DRAM is now free. This flag is used by the PCU and is available as an interrupt to the CPU.

35   22.2   STORAGE REQUIREMENTS FOR DECOMPRESSED CONTONE DATA IN DRAM

A single SoPEC must support a page of A4 length (11.7 inches) and Letter width (8.5 inches) at a resolution of 267 ppi in 4 colors and a print speed of 1 side per 2 seconds. The printheads specified in the Bi-lithic Printhead Specification [2] have 13824 nozzles per color to provide full

bleed printing for A4 and Letter. At 267 ppi, there are 2304 contone pixels[9] per line represented by 288 JPEG blocks per color. However each of these blocks actually stores data for 8 lines, since a single JPEG block is 8 x 8 pixels. The CDU produces contone data for 8 lines in parallel, while the HCU processes data linearly across a line on a line by line basis. The contone data is decoded only once and then buffered in DRAM. This means we require two sets of 8 buffer-lines - one set of 8 buffer lines is being consumed by the CFU while the other set of 8 buffer lines is being generated by the CDU.

The buffer requirement can be reduced by using a 1.5 buffering scheme, where the CDU fills 8 lines while the CFU consumes 4 lines. The buffer space required is a minimum of 12 line stores per color, for a total space of 108 KBytes[10]. A circular buffer scheme is employed whereby the CDU may only begin to write a line of JPEG blocks (equals 8 lines of contone data) when there are 8-lines free in the buffer. Once the full 8 lines have been written by the CDU, the CFU may now begin to read them on a line by line basis.

This reduction in buffering comes with the cost of an increased peak bandwidth requirement for the CDU write access to DRAM. The CDU must be able to write the decompressed contone at twice the rate at which the CFU reads the data. To allow for trade-offs to be made between peak bandwidth and amount of storage, the size of the circular buffer is configurable. For example, if the circular buffer is configured to be 16 lines it behaves like a double-buffer scheme where the peak bandwidth requirements of the CDU and CFU are equal. An increase over 16 lines allows the CDU to write ahead of the CFU and provides it with a margin to cope with very poor local compression ratios in the image.

SoPEC should also provide support for A3 printing and printing at resolutions above 267 ppi. This increases the storage requirement for the decompressed contone data (buffer) in DRAM. Table 143 gives the storage requirements for the decompressed contone data at some sample contone resolutions for different page sizes. It assumes 4 color planes of contone data and a 1.5 buffering scheme.

Table 143. Storage requirements for decompressed contone data (buffer)

Page size	Contone resolution (ppi)	Scale factor[a]	Pixels per line	Storage required (kBytes)
A4/Letter[b]	267	6	2304	108[d]
	400	4	3456	162
	800	2	6912	324
A3[c]	267	6	3248	152.25
	400	4	4872	228.37
	800	2	9744	456.75

[9]Pixels may be 8, 16, 24 or 32 bits depending on the number of color planes (8-bits per color)

[10]12 lines x 4 colors x 2304 bytes (assumes 267 ppi, 4 color, full bleed A4/Letter)

a. Required for CFU to convert to final output at 1600 dpi

b. Bi-lithic printhead has 13824 nozzles per color providing full bleed printing for A4/Letter

c. Bi-lithic printhead has 19488 nozzles per color providing full bleed printing for A3

d. 12 lines x 4 colors x 2304 bytes.

22.3 DECOMPRESSION PERFORMANCE REQUIREMENTS

The JPEG decoder core can produce a single color pixel every system clock (*pclk*) cycle, making it capable of decoding at a peak output rate of 8 bits/cycle. SoPEC processes 1 dot (bi-level in 6 colors) per system clock cycle to achieve a print speed of 1 side per 2 seconds for full bleed A4/Letter printing. The CFU replicates pixels a scale factor (SF) number of times in both the horizontal and vertical directions to convert the final output to 1600 ppi. Thus the CFU consumes a 4 color pixel (32 bits) every SF x SF cycles. The 1.5 buffering scheme described in section 22.2 on page 327 means that the CDU must write the data at twice this rate. With support for 4 colors at 267 ppi, the decompression output bandwidth requirement is 1.78 bits/cycle[11].

The JPEG decoder is fed directly from the main memory via the DRAM interface. The amount of compression determines the input bandwidth requirements for the CDU. As the level of compression increases, the bandwidth decreases, but the quality of the final output image can also decrease. Although the average compression ratio for contone data is expected to be 10:1, the average bandwidth allocated to the CDU allows for a local minimum compression ratio of 5:1 over a single line of JPEG blocks. This equates to a peak input bandwidth requirement of 0.36 bits/cycle for 4 colors at 267 ppi, full bleed A4/Letter printing at 1 side per 2 seconds.

Table 144 gives the decompression output bandwidth requirements for different resolutions of contone data to meet a print speed of 1 side per 2 seconds. Higher resolution requires higher bandwidth and larger storage for decompressed contone data in DRAM. A resolution of 400 ppi contone data in 4 colors requires 4 bits/cycle[12], which is practical using a 1.5 buffering scheme. However, a resolution of 800 ppi would require a double buffering scheme (16 lines) so the CDU only has to match the CFU consumption rate. In this case the decompression output bandwidth requirement is 8 bits/cycle[13], the limiting factor being the output rate of the JPEG decoder core.

Table 144. CDU performance requirements for full bleed A4/Letter printing at 1 side per 2 seconds.

Contone	Scale	Decompression output bandwidth

---

[11] $2 \times ((4 \text{ colors} \times 8 \text{ bits}) / (6 \times 6 \text{ cycles})) = 1.78 \text{ bits/cycle}$

[12] $2 \times ((4 \text{ colors} \times 8 \text{ bits}) / (4 \times 4 \text{ cycles})) = 4 \text{ bits/cycle}$

[13] $(4 \text{ colors} \times 8 \text{ bits}) / (2 \times 2 \text{ cycles}) = 8 \text{ bits/cycle}$

resolution (ppi)	factor	requirement (bits/cycle)[a]
267	6	1.78
400	4	4
800	2	8[b]

a. Assumes 4 color pixel contone data and a 12 line buffer.

b. Scale factor 2 requires at least a 16 line buffer.

22.4   DATA FLOW

Figure 136 shows the general data flow for contone data - compressed contone planes are read from DRAM by the CDU, and the decompressed contone data is written to the 12-line circular buffer in DRAM. The line buffers are subsequently read by the CFU.

The CDU allows the contone data to be passed directly on, which will be the case if the color represented by each color plane in the JPEG image is an available ink. For example, the four colors may be C, M, Y, and K, directly represented by CMYK inks. The four colors may represent gold, metallic green etc. for multi-SoPEC printing with exact colors.

However JPEG produces better compression ratios for a given visible quality when luminance and chrominance channels are separated. With CMYK, K can be considered to be luminance, but C, M, and Y each contain luminance information, and so would need to be compressed with appropriate luminance tables. We therefore provide the means by which CMY can be passed to SoPEC as YCrCb. K does not need color conversion. When being JPEG compressed, CMY is typically converted to RGB, then to YCrCb and then finally JPEG compressed. At decompression, the YCrCb data is obtained and written to the decompressed contone store by the CDU. This is read by the CFU where the YCrCb can then be optionally color converted to RGB, and finally back to CMY.

The external RIP provides conversion from RGB to YCrCb, specifically to match the actual hardware implementation of the inverse transform within SoPEC, as per CCIR 601-2 [24] except that Y, Cr and Cb are normalized to occupy all 256 levels of an 8-bit binary encoding.

The CFU provides the translation to either RGB or CMY. RGB is included since it is a necessary step to produce CMY, and some printers increase their color gamut by including RGB inks as well as CMYK.

22.5   IMPLEMENTATION

A block diagram of the CDU is shown in Figure 137.

All output signals from the CDU (*cdu_cfu_wradv8line*, *cdu_finishedband*, *cdu_icu_jpegerror*, and control signals to the DIU) must always be valid after reset. If the CDU is not currently decoding, *cdu_cfu_wradv8line*, *cdu_finishedband* and *cdu_icu_jpegerror* will always be 0.

The read control unit is responsible for keeping the JPEG decoder's input FIFO full by reading compressed contone bytestream from external DRAM via the DIU, and produces the *cdu_finishedband* signal. The write control unit accepts the output from the JPEG decoder a half JPEG block (32 bytes) at a time, writes it into a double-buffer, and writes the double buffered

decompressed half blocks to DRAM via the DIU, interacting with the CFU in order to share DRAM buffers.

22.5.1 Definitions of I/O

Table 145. CDU port list and description

Port name	Pins	I/O	Description
Clocks and reset			
Pclk	1	In	System clock.
Jclk	1	In	Gated version of system clock used to clock the JPEG decoder core and logic at the output of the core. Allows for stalling of the JPEG core at a pixel sample boundary.
jclk_enable	1	Out	Gating signal for jclk.
prst_n	1	In	System reset, synchronous active low.
jrst_n	1	In	Reset for jclk domain, synchronous active low.
PCU interface			
pcu_cdu_sel	1	In	Block select from the PCU. When *pcu_cdu_sel* is high both *pcu_adr* and *pcu_dataout* are valid.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_adr[7:2]	6	In	PCU address bus. Only 6 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
cdu_pcu_rdy	1	Out	Ready signal to the PCU. When *cdu_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *cdu_pcu_datain* is valid.
cdu_pcu_datain[31:0]	32	Out	Read data bus to the PCU.
DIU read interface			
cdu_diu_rreq	1	Out	CDU read request, active high. A read request must be accompanied by a valid read address.
diu_cdu_rack	1	In	Acknowledge from DIU, active high. Indicates that a read request has been accepted and the new read address can be placed on the address bus, *cdu_diu_radr*.
cdu_diu_radr[21:5]	17	Out	CDU read address. 17 bits wide (256-bit aligned word).
diu_cdu_rvalid	1	In	Read data valid, active high. Indicates that valid read data is now on the read data bus, *diu_data*.

diu_data[63:0]	64	In	Read data from DRAM.
DIU write interface			
cdu_diu_wreq	1	Out	CDU write request, active high. A write request must be accompanied by a valid write address and valid write data.
diu_cdu_wack	1	In	Acknowledge from DIU, active high. Indicates that a write request has been accepted and the new write address can be placed on the address bus, *cdu_diu_wadr*.
cdu_diu_wadr[21:3]	19	Out	CDU write address. 19 bits wide (64-bit aligned word).
cdu_diu_wvalid	1	Out	Write data valid, active high. Indicates that valid data is now on the write data bus, *cdu_diu_data*.
cdu_diu_data[63:0]	64	Out	Write data bus.
CFU interface			
cfu_cdu_rdadvline	1	In	Read line pulse, active high. Indicates that the CFU has finished reading a line of decompressed contone data to the circular buffer in DRAM and that line of the buffer is now free.
cdu_cfu_linestore_rdy	1	Out	Indicates if the contone line store has 1 or more lines available to read by the CFU.
TE and LBD interface			
cdu_start_of_bandstore[21:5]	17	Out	Points to the 256-bit word that defines the start of the memory area allocated for page bands.
cdu_end_of_bandstore[21:5]	17	Out	Points to the 256-bit word that defines the last address of the memory area allocated for page bands.
ICU interface			
cdu_finishedband	1	Out	CDU's *finishedBand* flag, active high. Interrupt to the CPU to indicate that the CDU has finished processing a band of compressed contone data in DRAM and that area of DRAM is now free.This signal goes to both the interrupt controller and the PCU.
cdu_icu_jpegerror	1	Out	Active high interrupt indicating an error has occurred in the JPEG decoding process and decompression has stopped. A reset of the CDU must be performed to clear this interrupt.

22.5.2   Configuration registers

The configuration registers in the CDU are programmed via the PCU interface. Refer to section 21.8.2 on page 321 for the description of the protocol and timing diagrams for reading and writing registers in the CDU. Note that since addresses in SoPEC are byte aligned and the PCU only supports 32-bit register reads and writes, the lower 2 bits of the PCU address bus are not required to decode the address space for the CDU. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of cdu_pcu_datain.

Since the CDU, LBD and TE all access the page band store, they share two registers that enable sequential memory accesses to the page band stores to be circular in nature. Table 146 lists these two registers.

Table 146. Registers shared between the CDU, LBD, and TE

Address (CDU_base+)	Register name	#bits	Value on reset	description
Setup registers (remain constant during the processing of multiple bands)				
0x80	StartOfBandStore[21:5]	17	0x0_0000	Points to the 256-bit word that defines the start of the memory area allocated for page bands. Circular address generation wraps to this start address.
0x84	EndOfBandStore[21:5]	17	0x1_3FFF	Points to the 256-bit word that defines the last address of the memory area allocated for page bands. If the current read address is from this address, then instead of adding 1 to the current address, the current address will be loaded from the StartOfBandStore register.

The software reset logic should include a circuit to ensure that both the pclk and jclk domains are reset regardless of the state of the jclk_enable when the reset is initiated.

The CDU contains the following additional registers:

Table 147. CDU registers

Address (CDU_base+)	Register name	#bits	Value on reset	Description
Control registers				
0x00	Reset	1	0x1	A write to this register causes a reset of the CDU. This terminates all internal operations within the CS6150. All

				configuration data previously loaded into the core except for the tables is deleted.
0x04	Go	1	0x0	Writing 1 to this register starts the CDU. Writing 0 to this register halts the CDU. When *Go* is deasserted the state-machines go to their idle states but all counters and configuration registers keep their values. When *Go* is asserted all counters are reset, but configuration registers keep their values (i.e. they don't get reset). *NextBandEnable* is cleared when *Go* is asserted. The CFU must be started before the CDU is started. *Go* must remain low for at least 384 *jclk* cycles after a hardware reset (*prst_n* = 0) to allow the JPEG core to complete its memory itnitialisation sequence. This register can be read to determine if the CDU is running (1 - running, 0 - stopped).
Setup registers				
0x0C	NumLinesAvail	7	0x0	The number of image lines of data that there is space available for in the decompressed data buffer in DRAM. If this drops < 8 the CDU will stall. In normal operation this value will start off atNumBuffLines and will be decremented by 8 whenever the CDU writes a line of JPEG blocks (8 lines of data) to DRAM and incremented by 1 whenever the CFU reads a line of data from DRAM. NumLinesAvail can be overwritten by the CPU to prevent the CDU from stalling.
0x10	MaxPlane	2	0x0	Defines the number of contone planes - 1.

334

				For example, this will be 0 for K (greyscale printing), 2 for CMY, and 3 for CMYK.
0x14	MaxBlock	13	0x000	Number of JPEG MCUs (or JPEG block equivalents, i.e. 8x8 bytes) in a line - 1.
0x18	BuffStartAdr[21:7]	15	0x0000	Points to the start of the decompressed contone circular buffer in DRAM, aligned to a half JPEG block boundary. A half JPEG block consists of 4 words of 256-bits, enough to hold 32 contone pixels in 4 colors, i.e. half a JPEG block.
0x1C	BuffEndAdr[21:7]	15	0x0000	Points to the start of the last half JPEG block at the end of the decompressed contone circular buffer in DRAM, aligned to a half JPEG block boundary. A half JPEG block consists of 4 words of 256-bits, enough to hold 32 contone pixels in 4 colors, i.e. half a JPEG block.
0x20	NumBuffLines[6:2]	5	0x03	Defines size of buffer in DRAM in terms of the number of decompressed contone lines. The size of the buffer should be a multiple of 4 lines with a minimum size of 8 lines.
0x24	BypassJpg	1	0x0	Determines whether or not the JPEG decoder will be bypassed (and hence pixels are copied directly from input to output) 0 - don't bypass, 1 - bypass Should not be changed between bands.
0x30	NextBandCurr-SourceAdr[21:5]	17	0x0_0000	The 256-bit aligned word address containing the start of the next band of compressed contone data in DRAM. This value is copied to *CurrSourceAdr* when both *DoneBand* is 1 and *NextBandEnable* is 1, or when *Go* transitions from 0 to 1.
0x34	NextBandEnd-	19	0x0_0000	The 64-bit aligned word address

	SourceAdr[21:3]			containing the last bytes of the next band of compressed contone data in DRAM. This value is copied to *EndSourceAdr* when when both *DoneBand* is 1 and *NextBandEnable* is 1, or when *Go* transitions from 0 to 1.
0x38	NextBandValid-BytesLastFetch	3	0x0	Indicates the number of valid bytes - 1 in the last 64-bit fetch of the next band of compressed contone data from DRAM. eg 0 implies bits 7:0 are valid, 1 implies bits 15:0 are valid, 7 implies all 63:0 bits are valid etc. This value is copied to *ValidBytesLastFetch* when both *DoneBand* is 1 and *NextBandEnable* is 1, or when *Go* transitions from 0 to 1.
0x3C	NextBandEnable	1	0x0	When *NextBandEnable* is 1 and *DoneBand* is 1 -*NextBandCurrSourceAdr* is copied to *CurrSourceAdr*, -*NextBandEndSourceAdr* is copied to *EndSourceAdr* -*NextBandValidBytesLastFetch* is copied to *ValidBytesLastFetch* -*DoneBand* is cleared, -*NextBandEnable* is cleared. *NextBandEnable* is cleared when *Go* is asserted. Note that *DoneBand* gets cleared regardless of the state of *Go*.
Read-only registers				
0x40	DoneBand	1	0x0	Specifies whether or not the current band has finished loading into the local FIFO. It is cleared to 0 when *Go* transitions from 0 to 1. When the last of the compressed contone data for the band has been loaded into the local FIFO, the *cdu_finishedband* signal is given out

				and the *DoneBand* flag is set. If *NextBandEnable* is 1 at this time then *CurrSourceAdr, EndSourceAdr* and *ValidBytesLastFetch* are updated with the values for the next band and *DoneBand* is cleared. Processing of the next band starts immediately. If *NextBandEnable* is 0 then the remainder of the CDU will continue to run, decompressing the data already loaded, while the read control unit waits for *NextBandEnable* to be set before it restarts.
0x44	CurrSourceAdr[21 :5]	17	0x0_0000	The current 256-bit aligned word address within the current band of compressed contone data in DRAM.
0x48	EndSourceAdr[21 :3]	19	0x0_0000	The 64-bit aligned word address containing the last bytes of the current band of compressed contone data in DRAM.
0x4C	ValidBytesLastFet ch	3	0x00	Indicates the number of valid bytes - 1 in the last 64-bit fetch of the current band of compressed contone data from DRAM. eg 0 implies bits 7:0 are valid, 1 implies bits 15:0 are valid, 7 implies all 63:0 bits are valid etc.
JPEG decoder core setup registers				
0x50	JpgDecMask	5	0x00	As segments are decoded they can also be output on the *DecJpg* (*JpgDecHdr*) port with the user selecting the segments for output by setting bits in the jpgDecMask port as follows: 4 SOF+SOS+DNL 3 COM+APP 2 DRI 1 DQT 0 DHT If any one of the bits of jpgDecMask is

337

				asserted then the SOI and EOI markers are also passed to the *DecJpg* port.
0x54	JpgDecTType	1	0x0	Test type selector: 0 - DCT coefficients displayed on *JpgDecTdata* 1 - QDCT coefficient displayed on *JpgDecTdata*
0x58	JpgDecTestEn	1	0x0	Signal which causes the memories to be bypassed for test purposes.
0x5C	JpgDecPType	4	0x0	Signal specifying parameters to be placed on port *JpgDecPValue* (See Table).
JPEG decoder core read-only status registers				
0x60	JpgDecHdr	8	0x00	Selected header segments from the JPEG stream that is currently being decoded. Segments selected using *JpgMask*.
0x64	JpgDecTData	13	0x0000	12 - TSOS output of CS1650, indicates the first output byte of the first 8×8 block of the test data. 11 - TSOB output of CS1650, indicates the first output byte of each 8×8 block of test data. 10-0 - 11-bit output test data port - displays DCT coefficients or quantized coefficients depending on value of *JpgDecTType*.
0x68	JpgDecPValue	16	0x0000	Decoding parameter bus which enables various parameters used by the core to be read. The data available on the PValue port is for information only, and does not contain control signals for the decoder core.
0x6C	JpgDecStatus	24	0x00_0000	Bit 23 - *jpg_core_stall* (if set, indicates that the JPEG core is stalled by gating of jclk as the output JPEG halfblock double-buffers of the CDU are full) Bit 22 - *pix_out_valid* (This signal is an output from the JPEG decoder core and

				is asserted when a pixel is being output Bits 21-16 - *fifo_contents* (Number of bytes in compressed contone FIFO at the input of CDU which feeds the JPEG decoder core) Bits 15-0 are JPEG decoder status outputs from the CS6150 (see Table for description of bits).

22.5.3 Typical operation

The CDU should only be started after the CFU has been started.

For the first band of data, users set up *NextBandCurrSourceAdr*, *NextBandEndSourceAdr*, *NextBandValidBytesLastFetch*, and the various *MaxPlane*, *MaxBlock*, *BuffStartBlockAdr*,

5   *BuffEndBlockAdr* and *NumBuffLines*. Users then set the CDU's *Go* bit to start processing of the band. When the compressed contone data for the band has finished being read in, the *cdu_finishedband* interrupt will be sent to the PCU and CPU indicating that the memory associated with the first band is now free. Processing can now start on the next band of contone data.

10   In order to process the next band *NextBandCurrSourceAdr*, *NextBandEndSourceAdr* and *NextBandValidBytesLastFetch* need to be updated before finally writing a 1 to *NextBandEnable*. There are 4 mechanisms for restarting the CDU between bands:

a. *cdu_finishedband* causes an interrupt to the CPU. The CDU will have set its *DoneBand* bit. The CPU reprograms the *NextBandCurrSourceAdr*, *NextBandEndSourceAdr* and
15   *NextBandValidBytesLastFetch* registers, and sets *NextBandEnable* to restart the CDU.

b. The CPU programs the CDU's *NextBandCurrSourceAdr*, *NextBandCurrEndAdr* and *Next-BandValidBytesLastFetch* registers and sets the *NextBandEnable* bit before the end of the current band. At the end of the current band the CDU sets *DoneBand*. As *NextBandEnable* is already 1, the CDU starts processing the next band immediately.

20   c. The PCU is programmed so that *cdu_finishedband* triggers the PCU to execute commands from DRAM to reprogram the *NextBandCurrSourceAdr*, *NextBandEndSourceAdr* and *Next-BandValidBytesLastFetch* registers and set the *NextBandEnable* bit to start the CDU processing the next band. The advantage of this scheme is that the CPU could process band headers in advance and store the band commands in DRAM ready for execution.

25   d.   This is a combination of *b* and *c* above. The PCU (rather than the CPU in *b*) programs the CDU's *NextBandCurrSourceAdr*, *NextBandCurrEndAdr* and *NextBandValidBytesLastFetch* registers and sets the *NextBandEnable* bit before the end of the current band. At the end of the current band the CDU sets *DoneBand* and pulses *cdu_finishedband*. As *NextBandEnable* is already 1, the CDU starts processing the next band immediately. Simultaneously,
30   *cdu_finishedband* triggers the PCU to fetch commands from DRAM. The CDU will have restarted by the time the PCU has fetched commands from DRAM. The PCU commands program the CDU's next band shadow registers and sets the *NextBandEnable* bit.

If an error occurs in the JPEG stream, the JPEG decoder will suspend its operation, an error bit will be set in the *JpgDecStatus* register and the core will ignore any input data and await a reset before starting decoding again. An interrupt is sent to the CPU by asserting *cdu_icu_jpegerror* and the CDU should then be reset by means of a write to its *Reset* register before a new page can be printed.

22.5.4   Read control unit

The read control unit is responsible for reading the compressed contone data and passing it to the JPEG decoder via the FIFO. The compressed contone data is read from DRAM in single 256-bit accesses, receiving the data from the DIU over 4 clock cycles (64-bits per cycle). The protocol and timing for read accesses to DRAM is described in section 20.9.1 on page 240. Read accesses to DRAM are implemented by means of the state machine described in Figure 138.

All counters and flags should be cleared after reset. When *Go* transitions from 0 to 1 all counters and flags should take their initial value. While the *Go* bit is set, the state machine relies on the *DoneBand* bit to tell it whether to attempt to read a band of compressed contone data. When *DoneBand* is set, the state machine does nothing. When *DoneBand* is clear, the state machine continues to load data into the JPEG input FIFO up to 256-bits at a time while there is space available in the FIFO. Note that the state machine has no knowledge about numbers of blocks or numbers of color planes - it merely keeps the JPEG input FIFO full by consecutive reads from DRAM. The DIU is responsible for ensuring that DRAM requests are satisfied at least at the peak DRAM read bandwidth of 0.36 bits/cycle (see section 22.3 on page 329).

A modulo 4 counter, *rd_count*, is use to count each of the 64-bits received in a 256-bit read access. It is incremented whenever *diu_cdu_rvalid* is asserted. As each 64-bit value is returned, indicated by *diu_cdu_rvalid* being asserted, *curr_source_adr* is compared to both *end_source_adr* and *end_of_bandstore*:

- If {*curr_source_adr,rd_count*} equals *end_source_adr*, the *end_of_band* control signal sent to the FIFO is 1 (to signify the end of the band), the *finishedCDUBand* signal is output, and the *DoneBand* bit is set. The remaining 64-bit values in the burst from the DIU are ignored, i.e. they are not written into the FIFO.

- If *rd_count equals 3 and {curr_source_adr,rd_count}* does not equal *end_source_adr*, then *curr_source_adr* is updated to be either *start_of_bandstore* or *curr_source_adr* + 1, depending on whether *curr_source_adr* also equals *end_of_bandstore*. The *end_of_band* control signal sent to the FIFO is 0.

curr_source_adr is output to the DIU as *cdu_diu_radr*.

A count is kept of the number of 64-bit values in the FIFO. When *diu_cdu_rvalid* is 1 and *ignore_data* is 0, data is written to the FIFO by asserting *FifoWr*, and *fifo_contents[3:0]* and *fifo_wr_adr[2:0]* are both incremented.

When *fifo_contents[3:0]* is greater than 0, *jpg_in_strb* is asserted to indicate that there is data available in the FIFO for the JPEG decoder core. The JPEG decoder core asserts *jpg_in_rdy* when it is ready to receive data from the FIFO. Note it is also possible to bypass the JPEG decoder core by setting the *BypassJpg* register to 1. In this case data is sent directly from the

FIFO to the half-block double-buffer. While the JPEG decoder is not stalled (*jpg_core_stall* equal 0), and *jpg_in_rdy* (or *bypass_jpg*) and *jpg_in_strb* are both 1, a byte of data is consumed by the JPEG decoder core. *fifo_rd_adr[5:0]* is then incremented to select the next byte. The read address is byte aligned, i.e. the upper 3 bits are input as the read address for the FIFO and the lower 3 bits are used to select a byte from the 64 bits. If *fifo_rd_adr[2:0]* = 111 then the next 64-bit value is read from the FIFO by asserting *fifo_rd,* and *fifo_contents[3:0]* is decremented.

22.5.5   Compressed contone FIFO

The compressed contone FIFO conceptually is a 64-bit input, and 8-bit output FIFO to account for the 64-bit data transfers from the DIU, and the 8-bit requirement of the JPEG decoder.

In reality, the FIFO is actually 8 entries deep and 65-bits wide (to accommodate two 256-bit accesses), with bits 63-0 carrying data, and bit 64 containing a 1-bit *end_of_band* flag. Whenever 64-bit data is written to the FIFO from the DIU, an *end_of_band* flag is also passed in from the read control unit. The *end_of_band* bit is 1 if this is the last data transfer for the current band, and 0 if it is not the last transfer. When *end_of_band* = 1 during an input, the *ValidBytesLastFetch* register is also copied to an image version of the same.

On the JPEG decoder side of the FIFO, the read address is byte aligned, i.e. the upper 3 bits are input as the read address for the FIFO and the lower 3 bits are used to select a byte from the 64 bits (1st byte corresponds to bits 7-0, second byte to bits 15-8 etc.). If bit 64 is set on the read, bits 63-0 contain the end of the bytestream for that band, and only the bytes specified by the image of *ValidBytesLastFetch* are valid bytes to be read and presented to the JPEG decoder. Note that *ValidBytesLastFetch* is copied to an image register as it may be possible for the CDU to be reprogrammed for the next band before the previous band's compressed contone data has been read from the FIFO (as an additional effect of this, the CDU has a non-problematic limitation in that each band of contone data must be more than 4 × 64-bits, or 32 bytes, in length).

22.5.6   CS6150 JPEG decoder

JPEG decoder functionality is implemented by means of a modified version of the Amphion CS6150 JPEG decoder core. The decoder is run at a nominal clock speed of 160 MHz. (Amphion have stated that the CS6150 JPEG decoder core can run at 185 MHz in 0.13um technology). The core is clocked by *jclk* which a gated version of the system clock pclk. Gating the clock provides a mechanism for stalling the JPEG decoder on a single color pixel-by-pixel basis. Control of the flow of output data is also provided by the *PixOutEnab* input to the JPEG decoder. However, this only allows stalling of the output at a JPEG block-boundary and is insufficient for SoPEC. Thus gating of the clock is employed and *PixOutEnab* is instead tied high.

The CS6150 decoder automatically extracts all relevant parameters from the JPEG bytestream and uses them to control the decoding of the image. The JPEG bytestream contains data for the Huffman tables, quantization tables, restart interval definition and frame and scan headers. The decoder parses and checks the JPEG bytestream automatically detecting and processing all the JPEG marker segments. After identifying the JPEG segments the decoder re-directs the data to the appropriate units to be stored or processed as appropriate. Any errors detected in the

bytestream, apart from those in the entropy coded segments, are signalled and, if an error is found, the decoder stops reading the JPEG stream and waits to be reset.

JPEG images must have their data stored in interleaved format with no subsampling. Images longer than 65536 lines are allowed: these must have an initial imageHeight of 0. If the image has

5 a Define Number Lines (DNL) marker at the end (normally necessary for standard JPEG, but not necessary for SoPEC's version of the CS6150), it must be equal to the total image height mod 64k or an error will be generated.

See the CS6150 Databook [21] for more details on how the core is used, and for timing diagrams of the interfaces. Note that [21] does not describe the use of the DNL marker in images of more

10 than 64k lines length as this is a modification to the core.

The CS6150 decoder can be bypassed by setting the *BypassJpg* register. If this register is set, then the data read from DRAM must be in the same format as if it was produced by the JPEG decoder: 8x8 blocks of pixels in the correct color order. The data is uncompressed and is therefore lossless.

15 The following subsections describe the means by which the CS6150 internals can be made visible.

22.5.6.1 JPEG decoder reset

The JPEG decoder has 2 possible types of reset, an asynchronous reset and a synchronous clear. In SoPEC the asynchronous reset is connected to the hardware synchronous reset of the

20 CDU and can be activated by any hardware reset to SoPEC (either from external pin or from any of the wake-up sources, e.g. USB activity, Wake-up register timeout) or by resetting the PEP section (*ResetSection* register in the CPR block).

The synchronous clear is connected to the software reset of the CDU and can be activated by the low to high transition of the *Go* register, or a software reset via the *Reset* register.

25 The 2 types of reset differ, in that the asynchronous reset, resets the JPEG core and causes the core to enter a memory initialization sequence that takes 384 clock cycles to complete after the reset is deasserted. The synchronous clear resets the core, but leaves the memory as is. This has some implications for programming the CDU.

In general the CDU should not be started (i.e. setting *Go* to 1) until at least 384 cycles after a

30 hardware reset. If the CDU is started before then, the memory initialization sequence will be terminated leaving the JPEG core memory in an unknown state. This is allowed if the memory is to be initialized from the incoming JPEG stream.

22.5.6.2 JPEG decoder parameter bus

The decoding parameter bus *JpgDecPValue* is a 16-bit port used to output various parameters

35 extracted from the input data stream and currently used by the core. The 4-bit selector input (*JpgDecPType*) determines which internal parameters are displayed on the parameter bus as per Table 148. The data available on the *PValue* port does not contain control signals used by the CS6150.

Table 148. Parameter bus definitions

40

342

PType	Output orientation	PValue
0x0	FY[15:0]	FY: number of lines in frame
0x1	FX[15:0]	FX: number of columns in frame
0x2	00_YMCU[13:0]	YMCU: number of MCUs in Y direction of the current scan
0x3	00_XMCU[13:0]	XMCU: number of MCUs in X direction of the current scan
0x4	Cs0[7:0]_Tq0[1:0]_V0[2:0]_H0[2:0]	Cs0: identifier for the first scan component Tq0: quantization table identifier for the first scan component V0: vertical sampling factor for the first scan component. Values = 1-4 H0: horizontal sampling factor for the first scan component. Values = 1-4
0x5	Cs1[7:0]_Tq1[1:0]_V1[2:0]_H1[2:0]	Cs1, Tq1, V1 and H1 for the second scan component. V1, H1 undefined if NS<2
0x6	Cs2[7:0]_Tq2[1:0]_V2[2:0]_H2[2:0]	Cs2, Tq2, V2 and H2 for the second scan component. V2, H2 undefined if NS<3
0x7	Cs3[7:0]_Tq3[1:0]_V3[2:0]_H3[2:0]	Cs3, Tq3, V3 and H3 for the second scan component. V3, H3 undefined if NS<4
0x8	CsH[15:0]	CsH: no. of rows in current scan
0x9	CsV[15:0]	CsV: no. of columns in current scan
0xA	DRI[15:0]	DRI: restart interval
0xB	000_HMAX[2:0]_VMAX[2: 0]_MCUBLK[3:0]_NS[2:0]	HMAX: maximal horizontal sampling factor in frame VMAX: maximal vertical sampling factor in frame MCUBLK: number of blocks per MCU of the current scan, from 1 to 10 NS: number of scan components in current scan, 1-4

22.5.6.3 JPEG decoder status register

The status register flags indicate the current state of the CS6150 operation. When an error is detected during the decoding process, the decompression process in the JPEG decoder is suspended and an interrupt is sent to the CPU by asserting *cdu_icu_jpegerror* (generated from

5   *DecError*). The CPU can check the source of the error by reading the *JpgDecStatus register*. The CS6150 waits until a reset process is invoked by asserting the hard reset *prst_n* or by a soft reset of the CDU. The individual bits of *JpgDecStatus* are set to zero at reset and active high to indicate an error condition as defined in Table 149.

Note: A *DecHfError* will not block the input as the core will try to recover and produce the correct

10   amount of pixel data. The *DecHfError* is cleared automatically at the start of the next image and so no intervention is required from the user. If any of the other errors occur in the decode mode then, following the error cancellation, the core will discard all input data until the next Start Of Image (SOI) without triggering any more errors.

The progress of the decoding can be monitored by observing the values of *TblDef*, *IDctInProg*,

15   *DecInProg* and *JpgInProg*.

Table 149. JPEG decoder status register definitions

Bit	Name	Description
15 - 12	TblDef[7:4]	Indicates the number of Huffman tables defined, 1bit/table.
11 - 8	TblDef[3:0]	Indicates the number of quantization tables defined, 1bit/table.
7	DecHfError	Set when an undefined Huffman table symbol is referenced during decoding.
6	CtlError	Set when an invalid SOF parameter or an invalid SOS parameter is detected. Also set when there is a mismatch between the DNL segment input to the core and the number of lines in the input image which have already been decoded. *Note that SoPEC's implementation of the CS6150 does not require a final DNL when the initial setting for ImageHeight is 0. This is to allow images longer than 64k lines.*
5	HtError	Set when an invalid DHT segment is detected.
4	QtError	Set when an invalid DQT segment is detected.
3	DecError	Set when anything other than a JPEG marker is input. Set when any of DecFlags[6:4] are set. Set when any data other than the SOI marker is detected at the start of a stream. Set when any SOF marker is detected other than SOF0. Set if incomplete Huffman or quantization definition is detected.
2	IDctInProg	Set when IDCT starts processing first data of a scan. Cleared when IDCT has processed the last data of a scan.
1	DecInProg	For each scan this signal is asserted after the SigSOS (Start of Scan Segment) signal has been output from the core and is de-asserted when the decoding of a scan is complete. It indicates that the core is in the decoding state.
0	JpgInProg	Set when core starts to process input data (JpgIn) and de-asserted when decoding has been completed i.e. when the last pixel of last block of the image is output.

22.5.7   Half-block buffer interface

Since the CDU writes 256 bits (4 x 64 bits) to memory at a time, it requires a double-buffer of 2 x 256 bits at its output. This is implemented in an 8 x 64 bit FIFO. It is required to be able to stall the JPEG decoder core at its output on a half JPEG block boundary, i.e. after 32 pixels (8 bits per pixel). We provide a mechanism for stalling the JPEG decoder core by gating the clock to the core(with jclk_enable) when the FIFO is full. The output FIFO is responsible for providing two buffered half JPEG blocks to decouple JPEG decoding (read control unit) from writing those JPEG blocks to DRAM (write control unit). Data coming in is in 8-bit quantities but data going out is in 64-bit quantities for a single color plane.

22.5.8   Write control unit

A line of JPEG blocks in 4 colors, or 8 lines of decompressed contone data, is stored in DRAM with the memory arrangement as shown Figure 139.The arrangement is in order to optimize access for reads by writing the data so that 4 color components are stored together in each 256-bit DRAM word.

The CDU writes 8 lines of data in parallel but stores the first 4 lines and second 4 lines separately in DRAM. The write sequence for a single line of JPEG 8x8 blocks in 4 colors, as shown in Figure 139, is as follows below and corresponds to the order in which pixels are output from the JPEG decoder core:

```
        block 0, color 0, line 0 in word p bits 63-0, line 1 in
word p+1 bits 63-0,
                            line 2 in word p+2 bits 63-0, line
3 in word p+3 bits 63-0,

        block 0, color 0, line 4 in word q bits 63-0, line 5 in
word q+1 bits 63-0,
                            line 6 in word q+2 bits 63-0, line
7 in word q+3 bits 63-0,

        block 0, color 1, line 0 in word p bits 127-64, line 1 in
word p+1 bits 127-64,
                            line 2 in word p+2 bits 127-64,
line 3 in word p+3 bits 127-64,

        block 0, color 1, line 4 in word q bits 127-64, line 5 in
word q+1 bits 127-64,
                            line 6 in word q+2 bits 127-64,
line 7 in word q+3 bits 127-64,

        repeat for block 0 color 2, block 0 color 3........

        block 1, color 0, line 0 in word p+4 bits 63-0, line 1 in
word p+5 bits 63-0,

        etc.............................................

        block N, color 3, line 4 in word q+4n bits 255-192, line 5
in word q+4n+1 bits 255-192,
                            line 6 in word q+4n+2 bits 255-
192, line 7 in word q+4n+3 bit 255-192
```

In SoPEC data is written to DRAM 256 bits at a time. The DIU receives a 64-bit aligned address from the CDU, i.e. the lower 2 bits indicate which 64-bits within a 256-bit location are being written to. With that address the DIU also receives half a JPEG block (4 lines) in a single color, 4 x 64 bits

over 4 cycles. All accesses to DRAM must be padded to 256 bits or the bits which should not be written are masked using the individual bit write inputs of the DRAM. When writing decompressed contone data from the CDU, only 64 bits out of the 256-bit access to DRAM are valid, and the remaining bits of the write are masked by the DIU. This means that the decompressed contone data is written to DRAM in 4 back-to-back 64-bit write masked accesses to 4 consecutive 256-bit DRAM locations/words.

Writing of decompressed contone data to DRAM is implemented by the state machine in Figure 140. The CDU writes the decompressed contone data to DRAM half a JPEG block at a time, 4 x 64 bits over 4 cycles. All counters and flags should be cleared after reset. When *Go* transitions from 0 to 1 all counters and flags should take their initial value. While the *Go* bit is set, the state machine relies on the *half_block_ok_to_read* and *line_store_ok_to_write* flags to tell it whether to attempt to write a half JPEG block to DRAM. Once the half-block buffer interface contains a half JPEG block, the state machine requests a write access to DRAM by asserting *cdu_diu_wreq* and providing the write address, corresponding to the first 64-bit value to be written, on *cdu_diu_wadr* (only the address the first 64-bit value in each access of 4x64 bits is issued by the CDU. The DIU can generate the addresses for the second, third and fourth 64-bit values). The state machine then waits to receive an acknowledge from the DIU before initiating a read of 4 64-bit values from the half-block buffer interface by asserting *rd_adv* for 4 cycles. The output *cdu_diu_wvalid* is asserted in the cycle after *rd_adv* to indicate to the DIU that valid data is present on the *cdu_diu_data* bus and should be written to the specified address in DRAM. A *rd_adv_half_block* pulse is then sent to the half-block buffer interface to indicate that the current read buffer has been read and should now be available to be written to again. The state machine then returns to the request state.

The pseudocode below shows how the write address is calculated on a per clock cycle basis. Note counters and flags should be cleared after reset. When *Go* transitions from 0 to 1 all counters and flags should be cleared and *lwr_halfblock_adr* gets loaded with *buff_start_adr* and *upr_halfblock_adr* gets loaded with *buff_start_adr* + *max_block* + 1.

```
// assign write address output to DRAM
    cdu_diu_wadr[6:5]   = 00              // corresponds to
linenumber, only first address is
                                          // issued for each DRAM
access. Thus line is always 0.
                                          // The DIU generates these
bits of the address.
    cdu_diu_wadr[4:3]   = color

    if (half == 1) then
        cdu_diu_wadr[21:7] = upr_halfblock_adr      // for lines
4-7 of JPEG block
    else
```

346

```
        cdu_diu_wadr[21:7] = lwr_halfblock_adr        // for lines
0-3 of JPEG block


// update half, color, block and addresses after each DRAM
write access
    if (rd_adv_half_block == 1) then
        if (half == 1) then
            half = 0
            if (color == max_plane) then
                color = 0
                if (block == max_block) then        // end of writing
a line of JPEG blocks
                        pulse wradv8line
                        block = 0


                        // update half block address for start of next
line of JPEG blocks taking
                        // account of address wrapping in circular
buffer and 4 line offset
                        if (upr_halfblock_adr == buff_end_adr) then
                            upr_halfblock_adr  =  buff_start_adr   +
max_block + 1
                            elsif (upr_halfblock_adr + max_block + 1 ==
buff_end_adr) then
                            upr_halfblock_adr = buff_start_adr
                        else
                            upr_halfblock_adr  =  upr_halfblock_adr  +
max_block + 2
                else
                    block ++
                    upr_halfblock_adr ++            // move to address
for lines 4-7 for next block
            else
                color ++
        else
            half = 1
            if (color == max_plane) then
                if (block == max_block) then    // end of writing a
line of JPEG blocks


                    // update half block address for start of next
line of JPEG blocks taking
                    // account of address wrapping in circular
buffer and 4 line offset
                    if (lwr_halfblock_adr == buff_end_adr) then
```

347

```
                                        lwr_halfblock_adr    =    buff_start_adr    +
        max_block + 1
                            elsif  (lwr_halfblock_adr  +  max_block  +  1  ==
        buff_end_adr) then
                                    lwr_halfblock_adr = buff_start_adr
                            else
                                    lwr_halfblock_adr  =  lwr_halfblock_adr  +
        max_block + 2

                            else
                                    lwr_halfblock_adr ++           // move to address
        for lines 0-3 for next block
```

22.5.9   Contone line store interface

The contone line store interface is responsible for providing the control over the shared resource in DRAM. The CDU writes 8 lines of data in up to 4 color planes, and the CFU reads them line-at-a-time. The contone line store interface provides the mechanism for keeping track of the number of lines stored in DRAM, and provides signals so that a given line cannot be read from until the complete line has been written.

The CDU writes 8 lines of data in parallel but writes the first 4 lines and second 4 lines to separate areas in DRAM. Thus, when the CFU has read 4 lines from DRAM that area now becomes free for the CDU to write to. Thus the size of the line store in DRAM should be a multiple of 4 lines. The minimum size of the line store interface is 8 lines, providing a single buffer scheme. Typical sizes are 12 lines for a 1.5 buffer scheme while 16 lines provides a double-buffer scheme.

The size of the contone line store is defined by *num_buff_lines.* A count is kept of the number of lines stored in DRAM that are available to be written to. When *Go* transitions from 0 to 1, *NumLinesAvail* is set to the value of *num_buff_lines.* The CDU may only begin to write to DRAM as long as there is space available for 8 lines, indicated when the *line_store_ok_to_write* bit is set. When the CDU has finished writing 8 lines, the write control unit sends an *wradv8line* pulse to the contone line store interface, and *NumLinesAvail* is decremented by 8. The write control unit then waits for *line_store_ok_to_write* to be set again.

If the contone line store is not empty (has one or more lines available in it), the CDU will indicate to the CFU via the *cdu_cfu_linestore_rdy* signal. The *cdu_cfu_linestore_rdy* signal is generated by comparing the *NumLinesAvail* with the programmed *num_buff_lines.* As the CFU reads a line from the contone line store it will pulse the *rdadvline* to indicate that it has read a full line from the line store. *NumLinesAvail* is incremented by 1 on receiving a *rdadvline* pulse.

To enable running the CDU while the CFU is not running the *NumLinesAvail* register can also be updated via the configuration register interface. In this scenario the CPU polls the value of the *NumLinesAvail* register and overwrites it to prevent stalling of the CDU (*NumLinesAvail* < 8). The CPU will always have priority in any updating of the *NumLinesAvail* register.

23   Contone FIFO Unit (CFU)

23.1   OVERVIEW

The Contone FIFO Unit (CFU) is responsible for reading the decompressed contone data layer from the circular buffer in DRAM, performing optional color conversion from YCrCb to RGB followed by optional color inversion in up to 4 color planes, and then feeding the data on to the HCU. Scaling of data is performed in the horizontal and vertical directions by the CFU so that the output to the HCU matches the printer resolution. Non-integer scaling is supported in both the horizontal and vertical directions. Typically, the scale factor will be the same in both directions but may be programmed to be different.

23.2   BANDWIDTH REQUIREMENTS

The CFU must read the contone data from DRAM fast enough to match the rate at which the contone data is consumed by the HCU.

Pixels of contone data are replicated a X scale factor (SF) number of times in the X direction and Y scale factor (SF) number of times in the Y direction to convert the final output to 1600 dpi. Replication in the X direction is performed at the output of the CFU on a pixel-by-pixel basis while replication in the Y direction is performed by the CFU reading each line a number of times, according to the Y-scale factor, from DRAM. The HCU generates 1 dot (bi-level in 6 colors) per system clock cycle to achieve a print speed of 1 side per 2 seconds for full bleed A4/Letter printing. The CFU output buffer needs to be supplied with a 4 color contone pixel (32 bits) every SF cycles. With support for 4 colors at 267 ppi the CFU must read data from DRAM at 5.33 bits/cycle[14].

23.3   COLOR SPACE CONVERSION

The CFU allows the contone data to be passed directly on, which will be the case if the color represented by each color plane in the JPEG image is an available ink. For example, the four colors may be C, M, Y, and K, directly represented by CMYK inks. The four colors may represent gold, metallic green etc. for multi-SoPEC printing with exact colors.

JPEG produces better compression ratios for a given visible quality when luminance and chrominance channels are separated. With CMYK, K can be considered to be luminance, but C, M and Y each contain luminance information and so would need to be compressed with appropriate luminance tables. We therefore provide the means by which CMY can be passed to SoPEC as YCrCb. K does not need color conversion.

When being JPEG compressed, CMY is typically converted to RGB, then to YCrCb and then finally JPEG compressed. At decompression, the YCrCb data is obtained, then color converted to RGB, and finally back to CMY.

The external RIP provides conversion from RGB to YCrCb, specifically to match the actual hardware implementation of the inverse transform within SoPEC, as per CCIR 601-2 [24] except that Y, Cr and Cb are normalized to occupy all 256 levels of an 8-bit binary encoding.

---

[14]32 bits / 6 cycles = 5.33 bits/cycle

The CFU provides the translation to either RGB or CMY. RGB is included since it is a necessary step to produce CMY, and some printers increase their color gamut by including RGB inks as well as CMYK.

Consequently the JPEG stream in the color space convertor is one of:

5
- 1 color plane, no color space conversion
- 2 color planes, no color space conversion
- 3 color planes, no color space conversion
- 3 color planes YCrCb, conversion to RGB
- 4 color planes, no color space conversion

10
- 4 color planes YCrCbX, conversion of YCrCb to RGB, no color conversion of X

The YCrCb to RGB conversion is described in [14]. Note that if the data is non-compressed, there is no specific advantage in performing color conversion (although the CDU and CFU do permit it).

23.4   COLOR SPACE INVERSION

In addition to performing optional color conversion the CFU also provides for optional bit-wise inversion in

15
up to 4 color planes. This provides the means by which the conversion to CMY may be finalised, or to may be used to provide planar correlation of the dither matrices.

The RGB to CMY conversion is given by the relationship:

- $C = 255 - R$
- $M = 255 - G$

20
- $Y = 255 - B$

These relationships require the page RIP to calculate the RGB from CMY as follows:

- $R = 255 - C$
- $G = 255 - M$
- $B = 255 - Y$

25   23.5   SCALING

Scaling of pixel data is performed in the horizontal and vertical directions by the CFU so that the output to the HCU matches the printer resolution. The CFU supports non-integer scaling with the scale factor represented by a numerator and a denominator. Only scaling up of the pixel data is allowed, i.e. the numerator should be greater than or equal to the denominator. For example, to

30
scale up by a factor of two and a half, the numerator is programmed as 5 and the denominator programmed as 2.

Scaling is implemented using a counter as described in the pseudocode below. An advance pulse is generated to move to the next dot (x-scaling) or line (y-scaling).

35
```
if (count + denominator - numerator >= 0) then
    count = count + denominator - numerator
    advance = 1
else
    count = count + denominator
```
40
```
    advance = 0
```
23.6   LEAD-IN AND LEAD-OUT CLIPPING

The JPEG algorithm encodes data on a block by block basis, each block consists of 64 8-bit pixels (representing 8 rows each of 8 pixels). If the image is not a multiple of 8 pixels in X and Y then padding must be present. This padding (extra pixels) will be present after decoding of the . JPEG bytestream.

5 Extra padded lines in the Y direction (which may get scaled up in the CFU) will be ignored in the HCU through the setting of the *BottomMargin* register.

Extra padded pixels in the X direction must also be removed so that the contone layer is clipped to the target page as necessary.

In the case of a multi-SoPEC system, 2 SoPECs may be responsible for printing the same side of

10 a page, e.g. SoPEC #1 controls printing of the left side of the page and SoPEC #2 controls printing of the right side of the page and shown in Figure 141. The division of the contone layer between the 2 SoPECs may not fall on a 8 pixel (JPEG block) boundary. The JPEG block on the boundary of the 2 SoPECs (JPEG block n below) will be the last JPEG block in the line printed by SoPEC #1 and the first JPEG block in the line printed by SoPEC #2. Pixels in this JPEG block not

15 destined for SoPEC #1 are ignored by appropriately setting the *LeadOutClipNum*. Pixels in this JPEG block not destined for SoPEC #2 must be ignored at the beginning of each line. The number of pixels to be ignored at the start of each line is specified by the *LeadInClipNum* register. It may also be the case that the CDU writes out more JPEG blocks than is required to be read by the CFU, as shown for SoPEC #2 below. In this case the value of the *MaxBlock* register in the

20 CDU is set to correspond to JPEG block *m* but the value for the *MaxBlock* register in the CFU is. set to correspond to JPEG block *m-1*. Thus JPEG block *m* is not read in by the CFU.

Additional clipping on contone pixels is required when they are scaled up to the printer's resolution. The scaling of the first valid pixel in the line is controlled by setting the *XstartCount* register. The *HcuLineLength* register defines the size of the target page for the contone layer at

25 the printer's resolution and controls the scaling of the last valid pixel in a line sent to the HCU.

23.7   IMPLEMENTATION

Figure 142 shows a block diagram of the CFU.

23.7.1   Definitions of I/O

Table 150. CFU port list and description

30

Port Name	Pins	I/O	Description
Clocks and reset			
pclk	1	In	System clock
prst_n	1	In	System reset, synchronous active low.
PCU interface			
pcu_cfu_sel	1	In	Block select from the PCU. When *pcu_cfu_sel* is high both *pcu_adr* and *pcu_dataout* are valid.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_adr[6:2]	4	In	PCU address bus. Only 5 bits are required to

			decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
cfu_pcu_rdy	1	Out	Ready signal to the PCU. When *cfu_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on cfu_pcu_datain is valid.
cfu_pcu_datain[31:0]	32	Out	Read data bus to the PCU.
DIU interface			
cfu_diu_rreq	1	Out	CFU read request, active high. A read request must be accompanied by a valid read address.
diu_cfu_rack	1	In	Acknowledge from DIU, active high. Indicates that a read request has been accepted and the new read address can be placed on the address bus, *cfu_diu_radr*.
cfu_diu_radr[21:5]	17	Out	CFU read address. 17 bits wide (256-bit aligned word).
diu_cfu_rvalid	1	In	Read data valid, active high. Indicates that valid read data is now on the read data bus, *diu_data*.
diu_data[63:0]	64	In	Read data from DRAM.
CDU interface			
cdu_cfu_linestore_rdy	1	In	When high indicates that the contone line store has 1 or more lines available to be read by the CFU.
cfu_cdu_rdadvline	1	Out	Read line pulse, active high. Indicates that the CFU has finished reading a line of decompressed contone data to the circular buffer in DRAM and that line of the buffer is now free.
HCU interface			
hcu_cfu_advdot	1	In	Informs the CFU that the HCU has captured the pixel data on cfu_hcu_c[0-3]data lines and the CFU can now place the next pixel on the data lines.
cfu_hcu_avail	1	Out	Indicates valid data present on cfu_hcu_c[0-3]data lines.
cfu_hcu_c0data[7:0]	8	Out	Pixel of data in contone plane 0.
cfu_hcu_c1data[7:0]	8	Out	Pixel of data in contone plane 1.
cfu_hcu_c2data[7:0]	8	Out	Pixel of data in contone plane 2.
cfu_hcu_c3data[7:0]	8	Out	Pixel of data in contone plane 3.

23.7.2   Configuration registers

The configuration registers in the CFU are programmed via the PCU interface. Refer to section 21.8.2 on page 321 for the description of the protocol and timing diagrams for reading and writing registers in the CFU. Note that since addresses in SoPEC are byte aligned and the PCU only supports 32-bit register reads and writes, the lower 2 bits of the PCU address bus are not required to decode the address space for the CFU. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *cfu_pcu_datain*. The configuration registers of the CFU are listed in Table 151:

Table 151. CFU registers

Address (CFU_base +)	Register Name	#bits	Value on Reset	Description
Control registers				
0x00	Reset	1	0x1	A write to this register causes a reset of the CFU.
0x04	Go	1	0x0	Writing 1 to this register starts the CFU. Writing 0 to this register halts the CFU. When *Go* is deasserted the state-machines go to their idle states but all counters and configuration registers keep their values. When *Go* is asserted all counters are reset, but configuration registers keep their values (i.e. they don't get reset). The CFU must be started before the CDU is started. This register can be read to determine if the CFU is running (1 - running, 0 - stopped).
Setup registers				
0x10	MaxBlock	13	0x000	Number of JPEG MCUs (or JPEG block equivalents, i.e. 8x8 bytes) in a line - 1.
0x14	BuffStartAdr[21:7]	15	0x0000	Points to the start of the decompressed contone circular buffer in DRAM, aligned to a half JPEG block boundary. A half JPEG block consists of 4 words of 256-bits, enough to hold 32 contone pixels in 4 colors, i.e. half a JPEG

				block.
0x18	BuffEndAdr[21:7]	15	0x0000	Points to the end of the decompressed contone circular buffer in DRAM, aligned to a half JPEG block boundary (address is inclusive). A half JPEG block consists of 4 words of 256-bits, enough to hold 32 contone pixels in 4 colors, i.e. half a JPEG block.
0x1C	4LineOffset	13	0x0000	Defines the offset between the start of one 4 line store to the start of the next 4 line store - 1. In Figure n page394 on page **Error! Bookmark not defined.**, if *BufStartAdr* corresponds to line 0 block 0 then *BuffStartAdr* + *4LineOffset* corresponds to line 4 block 0. *4LineOffset* is specified in units of128 bytes, eg 0 - 128 bytes,1 - 256 bytes etc. This register is required in addition to *MaxBlock* as the number of JPEG blocks in a line required by the CFU may be different from the number of JPEG blocks in a line written by the CDU.
0x20	YCrCb2RGB	1	0x0	Set this bit to enable conversion from YCrCb to RGB. Should not be changed between bands.

0x24	InvertColorPlane	4	0x0	Set these bits to perform bit-wise inversion on a per color plane basis.
				bit0 - 1 invert color plane 0
				      - 0 do not convert
				bit1 - 1 invert color plane 1
				      - 0 do not convert
				bit2 - 1 invert color plane 2
				      - 0 do not convert
				bit3 - 1 invert color plane 3
				Should not be changed between bands.

0x28	HcuLineLength	16	0x0000	Number of contone pixels - 1 in a line (after scaling). Equals the number of hcu_cfu_dotadv pulses - 1 received from the HCU for each line of contone data.
0x2C	LeadInClipNum	3	0x0	Number of contone pixels to be ignored at the start of a line (from JPEG block 0 in a line). They are not passed to the output buffer to be scaled in the X direction.
0x30	LeadOutClipNum	3	0x0	Number of contone pixels to be ignored at the end of a line (from JPEG block MaxBlock in a line). They are not passed to the output buffer to be scaled in the X direction.
0x34	XstartCount	8	0x00	Value to be loaded at the start of every line into the counter used for scaling in the X direction. Used to control the scaling of the first pixel in a line to be sent to the HCU. This value will typically be zero, except in the case where a number of dots are clipped on the lead in to a line.
0x38	XscaleNum	8	0x01	Numerator of contone scale factor in X direction.
0x3C	XscaleDenom	8	0x01	Denominator of contone scale factor in X direction.
0x40	YscaleNum	8	0x01	Numerator of contone scale factor in Y direction.
0x44	YscaleDenom	8	0x01	Denominator of contone scale factor in Y direction.

23.7.3   Storage of decompressed contone data in DRAM

The CFU reads decompressed contone data from DRAM in single 256-bit accesses. JPEG blocks of decompressed contone data are stored in DRAM with the memory arrangement as shown The arrangement is in order to optimize access for reads by writing the data so that 4 color components are stored together in each 256-bit DRAM word. The means that the CFU reads 64-bits in 4 colors from a single line in each 256-bit DRAM access.

The CFU reads data line at a time in 4 colors from DRAM. The read sequence, as shown in Figure 143, is as follows:

355

```
line 0, block 0 in word p of DRAM
line 0, block 1 in word p+4 of DRAM
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
line 0, block n in word p+4n of DRAM
(repeat to read line a number of times according to scale
factor)


line 1, block 0 in word p+1 of DRAM
line 1, block 1 in word p+5 of DRAM
etc. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

The CFU reads a complete line in up to 4 colors a Y scale factor number of times from DRAM before it moves on to read the next. When the CFU has finished reading 4 lines of contone data that 4 line store becomes available for the CDU to write to.

23.7.4   Decompressed contone buffer

Since the CFU reads 256 bits (4 colors x 64 bits) from memory at a time, it requires storage of at least 2 x 256 bits at its input. To allow for all possible DIU stall conditions the input buffer is increased to 3 x 256 bits to meet the CFU target bandwidth requirements. The CFU receives the data from the DIU over 4 clock cycles (64-bits of a single color per cycle). It is implemented as 4 buffers. Each buffer conceptually is a 64-bit input and 8-bit output buffer to account for the 64-bit data transfers from the DIU, and the 8-bit output per color plane to the color space converter.

On the DRAM side, *wr_buff* indicates the current buffer within each triple-buffer that writes are to occur to. *wr_sel* selects which triple-buffer to write the 64 bits of data to when *wr_en* is asserted. On the color space converter side, *rd_buff* indicates the current buffer within each triple-buffer that reads are to occur from. When *rd_en* is asserted a byte is read from each of the triple-buffers in parallel. *rd_sel* is used to select a byte from the 64 bits (1st byte corresponds to bits 7-0, second byte to bits 15-8 etc.).

Due to the limitations of available register arrays in IBM technology, the decompressed contone buffer is implemented as a quadruple buffer. While this offers some benefits for the CFU it is not necessitated by the bandwidth requirements of the CFU.

23.7.5   Y-scaling control unit

The Y-scaling control unit is responsible for reading the decompressed contone data and passing it to the color space converter via the decompressed contone buffer. The decompressed contone data is read from DRAM in single 256-bit accesses, receiving the data from the DIU over 4 clock cycles (64-bits per cycle). The protocol and timing for read accesses to DRAM is described in section 20.9.1 on page 240. Read accesses to DRAM are implemented by means of the state machine described in Figure 144.

All counters and flags should be cleared after reset. When *Go* transitions from 0 to 1 all counters and flags should take their initial value. While the *Go* bit is set, the state machine relies on the *line8_ok_to_read* and *buff_ok_to_write* flags to tell it whether to attempt to read a line of compressed contone data from DRAM. When *line8_ok_to_read* is 0 the state machine does

nothing. When *line8_ok_to_read* is 1 the state machine continues to load data into the decompressed contone buffer up to 256-bits at a time while there is space available in the buffer. A bit is kept for the status of each 64-bit buffer: *buff_avail[0]* and *buff_avail[1]*. It also keeps a single bit (*rd_buff*) for the current buffer that reads are to occur from, and a single bit (*wr_buff*) for the current buffer that writes are to occur to.

buff_ok_to_write equals ~*buff_avail[wr_buff]*. When a *wr_adv_buff* pulse is received, *buff_avail[wr_buff]* is set, and *wr_buff* is inverted. Whenever *diu_cfu_rvalid* is asserted, *wr_en* is asserted to write the 64-bits of data from DRAM to the buffer selected by *wr_sel* and *wr_buff*.

buff_ok_to_read equals *buff_avail[rd_buff]*. If there is data available in the buffer and the output double-buffer has space available (*outbuff_ok_to_write* equals 1) then data is read from the buffer by asserting *rd_en* and *rd_sel* gets incremented to point to the next value. *wr_adv* is asserted in the following cycle to write the data to the output double-buffer of the CFU. When finished reading the buffer, *rd_sel* equals b111 and *rd_en* is asserted, *buff_avail[rd_buff]* is set, and *rd_buff* is inverted.

Each line is read a number of times from DRAM, according to the Y-scale factor, before the CFU moves on to start reading the next line of decompressed contone data. Scaling to the printhead resolution in the Y direction is thus performed.

The pseudocode below shows how the read address from DRAM is calculated on a per clock cycle basis. Note all counters and flags should be cleared after reset or when *Go* is cleared. When a 1 is written to *Go*, both *curr_halfblock* and *line_start_halfblock* get loaded with *buff_start_adr*, and *y_scale_count* gets loaded with *y_scale_denom*. Scaling in the Y direction is implemented by line replication by re-reading lines from DRAM. The algorithm for non-integer scaling is described in the pseudocode below.

```
// assign read address output to DRAM
   cdu_diu_wadr[21:7] = curr_halfblock
   cdu_diu_wadr[6:5]  = line[1:0]

// update block, line, y_scale_count and addresses after
each DRAM read access
   if (wr_adv_buff == 1) then
      if (block == max_block) then   // end of reading a line
of contone in up to 4 colors
         block = 0
         // check whether to advance to next line of contone
data in DRAM
         if (y_scale_count + y_scale_denom - y_scale_num >= 0)
then
                y_scale_count = y_scale_count + y_scale_denom -
y_scale_num
```

```
                    pulse RdAdvline
                    if (line == 3) then        // end of reading 4 line
            store of contone data
                            line = 0
                            // update half block address for start of next
            line taking account of
                            // address wrapping in circular buffer and 4
            line offset
                            if (curr_halfblock == buff_end_adr) then
                                    curr_halfblock = buff_start_adr
                                    line_start_adr = buff_start_adr
                            elsif ((line_start_adr + 4line_offset) ==
            buff_end_adr)) then
                                    curr_halfblock = buff_start_adr
                                    line_start_adr = buff_start_adr
                            else
                                    curr_halfblock     =     line_start_adr     +
            4line_offset
                                    line_start_adr     =     line_start_adr     +
            4line_offset
                        else
                            line ++
                            curr_halfblock = line_start_adr
                    else
                        // re-read current line from DRAM
                        y_scale_count = y_scale_count + y_scale_denom
                        curr_halfblock = line_start_adr
                else
                    block ++
                    curr_halfblock ++
```

23.7.6   Contone line store interface

The contone line store interface is responsible for providing the control over the shared resource in DRAM. The CDU writes 8 lines of data in up to 4 color planes, and the CFU reads them line-at-a-time. The contone line store interface provides the mechanism for keeping track of the number of lines stored in DRAM, and provides signals so that a given line cannot be read from until the complete line has been written.

A count is kept of the number of lines that have been written to DRAM by the CDU and are available to be read by the CFU. At start-up, *buff_lines_avail* is set to the 0. The CFU may only begin to read from DRAM when the CDU has written 8 complete lines of contone data. When the CDU has finished writing 8 lines, it sends an *cdu_cfu_wradv8line* pulse to the CFU, and *buff_lines_avail* is incremented by 8. The CFU may continue reading from DRAM as long as *buff_lines_avail* is greater than 0. *line8_ok_to_read* is set while *buff_lines_avail* is greater than 0. When it has completely finished reading a line of contone data from DRAM, the Y-scaling control

358

unit sends a *RdAdvLine* signal to contone line store interface and to the CDU to free up the line in the buffer in DRAM. *buff_lines_avail* is decremented by 1 on receiving a *RdAdvline* pulse.

23.7.7 Color Space Converter (CSC)

The color space converter consists of 2 stages: optional color conversion from YCrCb to RGB followed by optional bit-wise inversion in up to 4 color planes.

The convert YCrCb to RGB block takes 3 8-bit inputs defined as Y, Cr, and Cb and outputs either the same data YCrCb or RGB. The YCrCb2RGB parameter is set to enable the conversion step from YCrCb to RGB. If YCrCb2RGB equals 0, the conversion does not take place, and the input pixels are passed to the second stage. The 4th color plane, if present, bypasses the convert YCrCb to RGB block. Note that the latency of the convert YCrCb to RGB block is 1 cycle. This latency should be equalized for the 4th color plane as it bypasses the block.

The second stage involves optional bit-wise inversion on a per color plane basis under the control of *invert_color_plane*. For example if the input is YCrCbK, then *YCrCb2RGB* can be set to 1 to convert YCrCb to RGB, and *invert_color_plane* can be set to 0111 to then convert the RGB to CMY, leaving K unchanged.

If *YCrCb2RGB* equals 0 and *invert_color_plane* equals 0000, no color conversion or color inversion will take place, so the output pixels will be the same as the input pixels.

Figure 145 shows a block diagram of the color space converter.

The convert YCrCb to RGB block is an implementation of [14]. Although only 10 bits of coefficients are used (1 sign bit, 1 integer bit, 8 fractional bits), full internal accuracy is maintained with 18 bits. The conversion is implemented as follows:

- $R^* = Y + (359/256)(Cr\text{-}128)$
- $G^* = Y - (183/256)(Cr\text{-}128) - (88/256)(Cb\text{-}128)$
- $B^* = Y + (454/256)(Cb\text{-}128)$

R^*, G^* and B^* are rounded to the nearest integer and saturated to the range 0-255 to give R, G and B. Note that, while a *Reset* results in all-zero output, a zero input gives output RGB = [0^{15}, 136^{16}, 0^{17}].

23.7.8 X-scaling control unit

The CFU has a 2 x 32-bit double-buffer at its output between the color space converter and the HCU. The X-scaling control unit performs the scaling of the contone data to the printers output resolution, provides the mechanism for keeping track of the current read and write buffers, and ensures that a buffer cannot be read from until it has been written to.

---

[15]-179 is saturated to 0

[16]135.5, with rounding becomes 136.

[17]-227 is saturated to 0

A bit is kept for the status of each 32-bit buffer: *buff_avail[0]* and *buff_avail[1]*. It also keeps a single bit (*rd_buff*) for the current buffer that reads are to occur from, and a single bit (*wr_buff*) for the current buffer that writes are to occur to.

The output value *outbuff_ok_to_write* equals *~buff_avail[wr_buff]*. Contone pixels are counted as they are received from the Y-scaling control unit, i.e. when *wr_adv* is 1. Pixels in the lead-in and lead-out areas are ignored, i.e. they are not written to the output buffer. Lead-in and lead-out clipping of pixels is implemented by the following pseudocode that generates the *wr_en* pulse for the output buffer.

```
if (wradv == 1) then
    if (pixel_count == {max_block,b111}) then
        pixel_count = 0
    else
        pixel_count ++
    if ((pixel_count < leadin_clip_num)
                OR  (pixel_count   >   ({max_block,b111}   -
leadout_clip_num))) then
        wr_en = 0
    else
        wr_en = 1
```

When a *wr_en* pulse is sent to the output double-buffer, *buff_avail[wr_buff]* is set, and *wr_buff* is inverted.

The output *cfu_hcu_avail* equals *buff_avail[rd_buff]*. When *cfu_hcu_avail* equals 1, this indicates to the HCU that data is available to be read from the CFU. The HCU responds by asserting *hcu_cfu_advdot* to indicate that the HCU has captured the pixel data on *cfu_hcu_c[0-3]data* lines and the CFU can now place the next pixel on the data lines.

The input pixels from the CSC may be scaled a non-integer number of times in the X direction to produce the output pixels for the HCU at the printhead resolution. Scaling is implemented by pixel replication. The algorithm for non-integer scaling is described in the pseudocode below. Note, *x_scale_count* should be loaded with *x_start_count* after reset and at the end of each line. This controls the amount by which the first pixel is scaled by. *hcu_line_length* and *hcu_cfu_dotadv* control the amount by which the last pixel in a line that is sent to the HCU is scaled by.

```
if (hcu_cfu_dotadv == 1) then
    if (x_scale_count + x_scale_denom - x_scale_num >= 0)
then
        x_scale_count  =  x_scale_count  +  x_scale_denom  -
x_scale_num
        rd_en = 1
    else
        x_scale_count = x_scale_count + x_scale_denom
        rd_en = 0
else
```

```
    x_scale_count = x_scale_count
    rd_en = 0
```

When a *rd_en* pulse is received, *buff_avail[rd_buff]* is cleared, and *rd_buff* is inverted.

A 16-bit counter, *dot_adv_count*, is used to keep a count of the number of *hcu_cfu_dotadv* pulses

5   received from the HCU. If the value of *dot_adv_count* equals *hcu_line_length* and a

hcu_cfu_dotadv pulse is received, then a *rd_en* pulse is genrated to present the next dot at the

output of the CFU, *dot_adv_count* is reset to 0 and *x_scale_count* is loaded with *x_start_count*.

24   Lossless Bi-level Decoder (LBD)

24.1   Overview

10   The Lossless Bi-level Decoder (LBD) is responsible for decompressing a single plane of bi-level
data. In SoPEC bi-level data is limited to a single spot color (typically black for text and line
graphics).

The input to the LBD is a single plane of bi-level data, read as a bitstream from DRAM. The LBD
is programmed with the start address of the compressed data, the length of the output

15   (decompressed) line, and the number of lines to decompress. Although the requirement for
SoPEC is to be able to print text at 10:1 compression, the LBD can cope with any compression
ratio if the requested DRAM access is available. A pass-through mode is provided for 1:1
compression. Ten-point plain text compresses with a ratio of about 50:1. Lossless bi-level
compression across an average page is about 20:1 with 10:1 possible for pages which compress

20   poorly.

The output of the LBD is a single plane of decompressed bi-level data. The decompressed bi-
level data is output to the SFU (Spot FIFO Unit), and in turn becomes an input to the HCU
(Halftoner/Compositor unit) for the next stage in the printing pipeline. The LBD also outputs a
lbd_finishedband control flag that is used by the PCU and is available as an interrupt to the CPU.

25   24.2   Main features of LBD

Figure 147 shows a schematic outline of the LBD and SFU.

The LBD is required to support compressed images of up to 800 dpi. If possible we would like to
support bi-level images of up to 1600 dpi. The line buffers must therefore be long enough to store
a complete line at 1600 dpi.

30   The PEC1 LBD is required to output 2 dots/cycle to the HCU. This throughput capability is
retained for SoPEC to minimise changes to the block, although in SoPEC the HCU will only read
1 dot/cycle. The PEC1 LDB outputs 16 bits in parallel to the PEC1 spot buffer. This is also
retained for SoPEC. Therefore the LBD in SoPEC can run much faster than is required. This is
useful for allowing stalls, e.g. due to band processing latency, to be absorbed.

35   The LBD has a pass through mode to cope with local negative compression. Pass through mode
is activated by a special run-length code. Pass through mode continues to either end of line or for
a pre-programmed number of bits, whichever is shorter. The special run-length code is always
executed as a run-length code, followed by pass through.

The LBD outputs decompressed bi-level data to the *NextLineFIFO* in the Spot FIFO Unit (SFU).

40   This stores the decompressed lines in DRAM, with a typical minimum of 2 lines stored in DRAM,

nominally 3 lines up to a programmable number of lines. The SFU's *NextLineFIFO* can fill while the SFU waits for write access to DRAM. Therefore the LBD must be able to support stalling at its output during a line.

The LBD uses the previous line in the decoding process. This is provided by the SFU via it's

5   *PrevLineFIFO*. Decoding can stall in the LBD while this FIFO waits to be filled from DRAM.

A signal *sfu_ldb_rdy* indicates that both the SFU's *NextLineFIFO* and *PrevLineFIFO* are available for writing and reading, respectively.

A configuration register in the LBD controls whether the first line being decoded at the start of a band uses the previous line read from the SFU or uses an all 0's line instead.

10   The line length is stored in DRAM must be programmable to a value greater than 128. An A4 line of 13824 dots requires 1.7Kbytes of storage. An A3 line of 19488 dots requires 2.4 Kbytes of storage.

The compressed spot data can be read at a rate of 1 bit/cycle for pass through mode 1:1 compression.

15   The LBD finished band signal is exported to the PCU and is additionally available to the CPU as an interrupt.

24.2.1   Bi-level Decoding in the LBD

The black bi-level layer is losslessly compressed using Silverbrook Modified Group 4 (SMG4) compression which is a version of Group 4 Facsimile compression [22] without Huffman and with

20   simplified run length encodings. The encoding are listed in Table 152 and Table 153.

Table 152. Bi-Level group 4 facsimile style compression encodings

	Encoding	Description
same as Group 4 Facsimile	1000	Pass Command: a0 ← b2, skip next two edges
	1	Vertical(0): a0 ← b1, color = !color
	110	Vertical(1): a0 ← b1 + 1, color = !color
	010	Vertical(-1): a0 ← b1 - 1, color = !color
	110000	Vertical(2): a0 ← b1 + 2, color = !color
	010000	Vertical(-2): a0 ← b1 - 2, color = !color
Unique to this implementation	100000	Vertical(3): a0 ← b1 + 3, color = !color
	000000	Vertical(-3): a0 ← b1 - 3, color = !color
	<RL><RL>100	Horizontal: a0 ← a0 + <RL> + <RL>

SMG4 has a pass through mode to cope with local negative compression. Pass through mode is activated by a special run-length code. Pass through mode continues to either end of line or for a

25   pre-programmed number of bits, whichever is shorter. The special run-length code is always executed as a run-length code, followed by pass through. The pass through escape code is a medium length run-length with a run of less than or equal to 31.

Table 153. Run length (RL) encodings

	Encoding	Description
Unique to this implementation	RRRRR1	Short Black Runlength (5 bits)
	RRRRR1	Short White Runlength (5 bits)
	RRRRRRRRR10	Medium Black Runlength (10 bits)
	RRRRRRRR10	Medium White Runlength (8 bits)
	RRRRRRRRR10	Medium Black Runlength with RRRRRRRRR <= 31, Enter pass through
	RRRRRRRR10	Medium White Runlength with RRRRRRRR <= 31, Enter pass through
	RRRRRRRRRRRRRR RR00	Long Black Runlength (15 bits)
	RRRRRRRRRRRRRR RR00	Long White Runlength (15 bits)

Since the compression is a bitstream, the encodings are read right (least significant bit) to left (most significant bit). The run lengths given as RRRRR in Table 153 are read in the same way

5   (least significant bit at the right to most significant bit at the left).

There is an additional enhancement to the G4 fax algorithm, it relates to pass through mode. It is possible for data to compress negatively using the G4 fax algorithm. On occasions like this it would be easier to pass the data to the LBD as un-compressed data. Pass through mode is a new feature that was not implemented in the PEC1 version of the LBD. When the LBD is in pass

10   through mode the least significant bit of the data stream is an un-compressed bit. This bit is used to construct the current line.

To enter pass through mode the LBD takes advantage of the way run lengths can be written. Usually if one of the runlength pair is less than or equal to 31 it should be encoded as a short runlength. However under the coding scheme of Table it is still legal to write it as a medium or

15   long runlength. The LBD has been designed so that if a short runlength value is detected in a medium runlength then once the horizontal command containing this runlength is decoded completely this will tell the LBD to enter pass through mode and the bits following the runlength is un-compressed data. The number of bits to pass through is either a programmed number of bits or the end of the line which ever comes first. Once the pass through mode is completed the

20   current color is the same as the color of the last bit of the passed through data.

24.2.2   DRAM Access Requirements

The compressed page store for contone, bi-level and raw tag data is 2 Mbytes. The LBD will access the compressed page store in single 256-bit DRAM reads. The LBD will need a 256-bit double buffer in its interface to the DIU. The LBD's DIU bandwidth requirements are summarized

25   in Table 154

Table 154. DRAM bandwidth requirements

Direction	Maximum number of cycles between each 256-bit DRAM access	Peak Bandwidth (bits/cycle)	Average Bandwidth (bits/cycle)
Read	2561 (1:1 compression)	1 (1:1 compression)	0.1 (10:1 compression)

1: At 1:1 compression the LBD requires 1 bit/cycle or 256 bits every 256 cycles.

5   24.3   IMPLEMENTATION

24.3.1   Definitions of IO

Table 155. LBD Port List

Port Name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	SoPEC Functional clock.
prst_n	1	In	Global reset signal.
Bandstore signals			
cdu_endofbandstore[21:5]	17	In	Address of the end of the current band of data. 256-bit word aligned DRAM address.
cdu_startofbandstore[21:5]	17	In	Address of the start of the current band of data. 256-bit word aligned DRAM address.
lbd_finishedband	1	Out	LBD finished band signal to PCU and Interrupt Controller.
DIU Interface signals			
lbd_diu_rreq	1	Out	LBD requests DRAM read. A read request must be accompanied by a valid read address.
lbd_diu_radr[21:5]	17	Out	Read address to DIU 17 bits wide (256-bit aligned word).
diu_lbd_rack	1	In	Acknowledge from DIU that read request has been accepted and new read address can be placed on *lbd_diu_radr*.
diu_data[63:0]	64	In	Data from DIU to SoPEC Units. First 64-bits is bits 63:0 of 256 bit word. Second 64-bits is bits 127:64 of 256 bit word.

			Third 64-bits is bits 191:128 of 256 bit word. Fourth 64-bits is bits 255:192 of 256 bit word.
diu_lbd_rvalid	1	In	Signal from DIU telling SoPEC Unit that valid read data is on the *diu_data* bus
PCU Interface data and control signals			
pcu_addr[5:2]	4	In	PCU address bus. Only 4 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
lbd_pcu_datain[31:0]	32	Out	Read data bus from the LBD to the PCU.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_lbd_sel	1	In	Block select from the PCU. When *pcu_lbd_sel* is high both *pcu_addr* and *pcu_dataout* are valid.
lbd_pcu_rdy	1	Out	Ready signal to the PCU. When *lbd_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *lbd_pcu_datain* is valid.
SFU Interface data and control signals			
sfu_lbd_rdy	1	In	Ready signal indicating SFU has previous line data available for reading and is also ready to be written to.
lbd_sfu_advline	1	Out	Advance line signal to previous and next line buffers
lbd_sfu_pladvword	1	Out	Advance word signal for previous line buffer.
sfu_lbd_pldata[15:0]	16	In	Data from the previous line buffer.
lbd_sfu_wdata[15:0]	16	Out	Write data for next line buffer.
lbd_sfu_wdatavalid	1	Out	Write data valid signal for next line buffer data.

24.3.2   Configuration Registers

Table 156. LBD Configuration Registers

Address (LBD_base +)	Register Name	#Bits	Value on Reset	Description
Control registers				
0x00	Reset	1	0x1	A write to this register causes a reset of the LBD. This register can be read to indicate the reset state: 0 - reset in progress 1 - reset not in progress
0x04	Go	1	0x0	Writing 1 to this register starts the LBD. Writing 0 to this register halts the LBD. The *Go* register is reset to 0 by the LBD when it finishes processing a band. When *Go* is deasserted the state-machines go to their idle states but all counters and configuration registers keep their values. When *Go* is asserted all counters are reset, but configuration registers keep their values (i.e. they don't get reset). The LBD should only be started after the SFU is started. This register can be read to determine if the LBD is running (1 - running, 0 - stopped).
Setup registers (constant for during processing the page)				
0x08	LineLength	16	0x0000	Width of expanded bi-level line (in dots) (must be set greater than 128 bits).
0x0C	PassThrough Enable	1	0x1	Writing 1 to this register enables passthrough mode. Writing 0 to this register disables pass-through mode thereby making the LBD compatible with PEC1.
0x10	PassThrough DotLength	16	0x0000	This is the dot length - 1 for which pass-through mode will last. If the end of the line is reached first then pass-through will be disabled. The value written to this register

				must be a non-zero value.
Work registers (need to be set up before processing a band)				
0x14	NextBandCurrReadAdr[21:5] (256-bit aligned DRAM address)	17	0x00000	Shadow register which is copied to *CurrReadAdr* when *(NextBandEnable == 1 & Go == 0)*. *NextBandCurrReadAdr* is the address of the start of the next band of compressed bi-level data in DRAM.
0x18	NextBandLinesRemaining	15	0x0000	Shadow register which is copied to *Lines-Remaining* when *(NextBandEnable == 1 & Go == 0)*. *NextBandLinesRemaining* is the number of lines to be decoded in the next band of compressed bi-level data.
0x1C	NextBandPrevLineSource	1	0x0	Shadow register which is copied to *Prev-LineSource* when *(NextBandEnable == 1 & Go == 0)*. 1 - use the previous line read from the SFU for decoding the first line at the start of the next band. 0 - ignore the previous line read from the SFU for decoding the first line at the start of the next band (an all 0's line is used instead).
0x20	NextBandEnable	1	0x0	If *(NextBandEnable == 1 & Go == 0)* then -*NextBandCurrReadAdr* is copied to *CurrReadAdr*, -*NextBandLinesRemaining* is copied to *LinesRemaining*, -*NextBandPrevLineSource* is copied to *PrevLineSource*, -*Go* is set, -*NextBandEnable* is cleared. To start LBD processing *NextBandEnable* should be set.
Work registers (read only for external access)				
0x24	CurrReadAdr [21:5] (256-bit	17	-	The current 256-bit aligned read address within the compressed bi-level image (DRAM address). Read only register.

	aligned DRAM address)			
0x28	LinesRemaining	15	-	Count of number of lines remaining to be decoded. The band has finished when this number reaches 0. Read only register.
0x2C	PrevLineSource	1	-	1 - uses the previous line read from the SFU for decoding the first line at the start of the next band. 0 - ignores the previous line read from the SFU for decoding the first line at the start of the next band (an all 0's line is used instead). Read only register.
0x30	CurrWriteAdr	15	-	The current dot position for writing to the SFU. Read only register.
0x34	FirstLineOfBand	1	-	Indicates whether the current line is considered to be the first line of the band. Read only register.

24.3.3 Starting the LBD between bands

The LBD should be started *after* the SFU. The LBD is programed with a start address for the compressed bi-level data, a decode line length, the source of the previous line and a count of how many lines to decode. The LBD's *NextBandEnable* bit should then be set (this will set LBD *Go*).

The LBD decodes a single band and then stops, clearing it's *Go* bit and issuing a pulse on *lbd_finishedband*. The LBD can then be restarted for the next band, while the HCU continues to process previously decoded bi-level data from the SFU.

There are 4 mechanisms for restarting the LBD between bands:

a. *lbd_finishedband* causes an interrupt to the CPU. The LBD will have stopped and cleared its *Go* bit. The CPU reprograms the LBD, typically the *NextBandCurrReadAdr*, *NextBandLinesRemaining* and *NextBandPrevLineSource* shadow registers, and sets *NextBandEnable* to restart the LBD.

b. The CPU programs the LBD's *NextBandCurrReadAdr*, *NextBandLinesRemaining*, and *NextBandPrevLineSource* shadow registers and sets the *NextBandEnable* flag before the end of the current band. At the end of the band the LBD clears *Go*, *NextBandEnable* is already set so the LBD restarts immediately.

c. The PCU is programmed so that *lbd_finishedband* triggers the PCU to execute commands from DRAM to reprogram the LBD's *NextBandCurrReadAdr*, *NextBandLinesRemaining*, and *NextBandPrevLineSource* shadow registers and set *NextBandEnable* to restart the LBD. The advantage of this scheme is that the CPU could process band headers in advance and store the band commands in DRAM ready for execution.

d.   This is a combination of *b* and *c* above. The PCU (rather than the CPU in *b*) programs the LBD's *NextBandCurrReadAdr*, *NextBandLinesRemaining,* and *NextBandPrevLineSource* shadow registers and sets the *NextBandEnable* flag before the end of the current band. At the end of the band the LBD clears *Go* and pulses *lbd_finishedband. NextBandEnable* is already set so the LBD restarts immediately. Simultaneously, *lbd_finishedband* triggers the PCU to fetch commands from DRAM. The LBD will have restarted by the time the PCU has fetched commands from DRAM. The PCU commands program the LBD's shadow registers and sets *NextBandEnable* for the next band.

24.3.4   Top-level Description

A block diagram of the LBD is shown in Figure 148.

The LBD contains the following sub-blocks:

Table 157. Functional sub-blocks in the LBD

name	Description
Registers and Resets	PCU interface and configuration registers. Also generates the *Go* and the *Reset* signals for the rest of the LBD
Stream Decoder	Accesses the bi-level description from the DRAM through the DIU interface. It decodes the bit stream into a command with arguments, which it then passes to the command controller.
Command Controller	Interprets the command from the stream decoder and provide the line fill unit with a limit address and color to fill the SFU Next Line Buffer. It also provides the next edge unit starting address to look for the next edge.
Next Edge Unit	Scans through the Previous Line Buffer using its current address to find the next edge of a color provided by the command controller. The next edge unit outputs this as the next current address back to the command controller and sets a valid bit when this address is at the next edge.
Line Fill Unit	Fills the SFU Next Line Buffer with a color from its current address up to a limit address. The color and limit are provided by the command controller.

In the following description the LBD decodes data for its current decode line but writes this data into the SFU's *next* line buffer.

Naming of signals and logical blocks are taken from [22].

The LBD is able to stall mid-line should the SFU be unable to supply a previous line or receive a current line frame due to band processing latency.

All output control signals from the LBD must always be valid after reset. For example, if the LBD is not currently decoding, *lbd_sfu_advline* (to the SFU) and *lbd_finishedband* will always be 0.

24.3.5   Registers and Resets sub-block description

Since the CDU, LBD and TE all access the page band store, they share two registers that enable sequential memory accesses to the page band stores to be circular in nature. The CDU chapter lists these two registers. The register descriptions for the LBD are listed in Table .

During initialisation of the LBD, the *LineLength* and the *LinesRemaining* configuration values are written to the LBD. The '*Registers and Resets*' sub-block supplies these signals to the other sub-blocks in the LBD. In the case of *LinesRemaining*, this number is decremented for every line that is completed by the LBD.

If pass through is used during a band the *PassThroughEnable* register needs to be programmed and *PassThroughDotLength* programmed with the length of the compressed bits in pass through mode.

PrevLineSource is programmed during the initialisation of a band, if the previous line supplied for the first line is a valid previous line, a 1 is written to *PrevLineSource* so that the data is used. If a 0 is written the LBD ignores the previous line information supplied and acts as if it is receiving all zeros for the previous line regardless of what the out of the SFU is.

The '*Registers and Resets*' sub-block also generates the resets used by the rest of the LBD and the *Go* bit which tells the LBD that it can start requesting data from the DIU and commence decoding of the compressed data stream.

24.3.6   Stream Decoder Sub-block Description

The Stream Decoder reads the compressed bi-level image from the DRAM via the DIU (single accesses of 256-bits) into a double 256-bit FIFO. The barrel shift register uses the 64-bit word from the FIFO to fill up the empty space created by the barrel shift register as it is shifting it's contents. The bit stream is decoded into a command/arguments pair, which in turn is passed to the command controller.

A dataflow block diagram of the stream decoder is shown in Figure 149.

24.3.6.1 *DecodeC - Decode Command*

The *DecodeC* logic encodes the command from bits 6..0 of the bit stream to output one of three commands: *SKIP*, *VERTICAL* and *RUNLENGTH*. It also provides an output to indicate how many bits were consumed, which feeds back to the barrel shift register.

There is a fourth command, *PASS_THROUGH*, which is not encoded in bits 6..0, instead it is inferred in a special runlength. If the stream decoder detects a short runlength value, i.e. a number less than 31, encoded as a medium runlength this tell the Stream Decoder that once the horizontal command containing this runlength is decoded completely the LBD enters *PASS_THROUGH* mode. Following the runlength there will be a number of bits that represent un-compressed data. The LBD will stay in *PASS_THROUGH* mode until all these bits have been decoded successfully, this will occur once a programmed number of bits is reached or the line ends, which ever comes first.

24.3.6.2 *DecodeD - Decode Delta*

The *DecodeD* logic decodes the run length from bits 20..3 of the bit stream. If *DecodeC* is decoding a vertical command, it will cause *DecodeD* to put constants of -3 through 3 on its output. The output *delta* is a 15 bit number, which is generally considered to be positive, but since it

370

needs to only address to 13824 dots for an A4 page and 19488 dots for an A3 page (of 32,768), a 2's complement representation of -3,-2,-1 will work correctly for the data pipeline that follows. This unit also outputs how many bits were consumed.

In the case of *PASS_THROUGH* mode, *DecodeD* parses the bits that represent the un-compressed data and this is used by the Line Fill Unit to construct the current line frame. *DecodeD* parses the bits at one bit per clock cycle and passes the bit in the less significant bit location of *delta* to the line fill unit.

DecodeD currently requires to know the color of the run length to decode it correctly as black and white runs are encoded differently. The stream decoder keeps track of the next color based on the current color and the current command.

24.3.6.3 State-machine

This state machine continuously fetches consecutive DRAM data whenever there is enough free space in the FIFO, thereby keeping the barrel shift register full so it can continually decode commands for the command controller. Note in Figure 149 that each read cycle *curr_read_addr* is compared to *end_of_band_store*. If the two are equal, *curr_read_addr* is loaded with *start_of_band_store* (circular memory addressing). Otherwise *curr_read_addr* is simply incremented. *start_of_band_store* and *end_of_band_store* need to be programed so that the distance between them is a multiple of the 256-bit DRAM word size.

When the state machine decodes a *SKIP* command, the state machine provides two *SKIP* instructions to the command controller.

The *RUNLENGTH* command has two different run lengths. The two run lengths are passed to the command controller as separate *RUNLENGTH* instructions. In the first instruction fetch, the first run length is passed, and the state machine selects the *DecodeD* shift value for the barrel shift. In the second instruction fetch from the command controller another *RUNLENGTH* instruction is generated and the respective shift value is decoded. This is achieved by forcing *DecodeC* to output a second *RUNLENGTH* instruction and the respective shift value is decoded.

For *PASS_THROUGH* mode, the *PASS_THROUGH* command is issued every time the command controller requests a new command. It does this until all the un-compressed bits have been processed.

24.3.7 Command Controller Sub-block Description

The Command Controller interprets the command from the Stream Decoder and provides the line fill unit with a limit address and color to fill the SFU Next Line Buffer. It provides the next edge unit with a starting address to look for the next edge and is responsible for detecting the end of line and generating the *eob_cc* signal that is passed to the line fill unit.

A dataflow block diagram of the command controller is shown in Figure 150. Note that data names such as *a0* and *b1p* are taken from [22], and they denote the reference or starting changing element on the coding line and the first changing element on the reference line to the right of *a0* and of the opposite color to *a0* respectively.

24.3.7.1 State machine

The following is an explanation of all the states that the state machine utilizes.

371

i   START

This is the state that the Command Controller enters when a hard or soft reset occurs or when *Go* has been de-asserted. This state cannot be left until the reset has been removed, *Go* has been asserted and the *NEU* (Next Edge Unit), the *SD* (Stream Decoder) and the SFU are ready.

5   *ii   AWAIT_BUFFER*

The *NEU* contains a buffer memory for the data it receives from the SFU. When the command controller enters this state the *NEU* detects this and starts buffering data, the command controller is able to leave this state when the state machine in the *NEU* has entered the *NEU_RUNNING* state. Once this occurs the command controller can proceed to the *PARSE* state.

10   *iii   PAUSE_CC*

During the decode of a line it is possible for the FIFO in the stream decoder to get starved of data if the DRAM is not able to supply replacement data fast enough. Additionally the SFU can also stall mid-line due to band processing latency. If either of these cases occurs the LBD needs to pause until the stream decoder gets more of the compressed data stream from the DRAM or the

15   SFU can receive or deliver new frames. All of the remaining states check if *sdvalid* goes to zero (this denotes a starving of the stream decoder) or if *sfu_lbd_rdy* goes to zero and that the LBD needs to pause. *PAUSE_CC* is the state that the command controller enters to achieve this and it does not leave this state until *sdvalid* and *sfu_lbd_rdy* are both asserted and the LBD can recommence decompressing.

20   *iv   PARSE*

Once the command controller enters the PARSE state it uses the information that is supplied by the stream decoder. The first clock cycle of the state sees the *sdack* signal getting asserted informing the stream decoder that the current register information is being used so that it can fetch the next command.

25   When in this state the command controller can receive one of four valid commands:

a) Runlength or Horizontal

For this command the value given as delta is an integer that denotes the number of bits of the current color that must be added to the current line.

Should the current line position, *a0*, be added to the delta and the result be greater than the final

30   position of the current frame being processed by the Line Fill Unit (only 16 bits at a time), it is necessary for the command controller to wait for the Line Fill Unit (LFU) to process up to that point. The command controller changes into the *WAIT_FOR_RUNLENGTH* state while this occurs.

When the current line position, *a0*, and the delta together equal or exceed the LINE_LENGTH,

35   which is programmed during initialisation, then this denotes that it is the end of the current line. The command controller signals this to the rest of the LBD and then returns to the *START* state.

b) Vertical

When this command is received, it tells the command controller that, in the previous line, it needs to find a change from the current color to opposite of the current color, i.e. if the current color is

40   white it looks from the current position in the previous line for the next time where there is a

change in color from white to black. It is important to note that if a black to white change occurs first it is ignored.

Once this edge has been detected, the delta will denote which of the vertical commands to use, refer to Table . The delta will denote where the changing element in the current line is relative to the changing element on the previous line, for a Vertical(2) the new changing element position in the current line will correspond to the two bits extra from changing element position in the previous line.

Should the next edge not be detected in the current frame under review in the *NEU*, then the command controller enters the *WAIT_FOR_NE* state and waits there until the next edge is found.

c) Skip

A skip follow the same functionality as to *Vertical*(0) commands but the color in the current line is not changed as it is been filled out. The stream decoder supplies what looks like two separate skip commands that the command controller treats the same a two *Vertical*(0) commands and has been coded not to change the current color in this case.

d) Pass Through

When in pass through mode the stream decoder supplies one bit per clock cycle that is uses to construct the current frame. Once pass through mode is completed, which is controlled in the stream decoder, the LBD can recommence normal decompression again. The current color after pass through mode is the same color as the last bit in un-compressed data stream. Pass through mode does not need an extra state in the command controller as each pass through command received from the stream decoder can always be processed in one clock cycle.

v   WAIT_FOR_RUNLENGTH

As some *RUNLENGTH's* can carry over more than one 16-bit frame, this means that the Line Fill Unit needs longer than one clock cycle to write out all the bits represented by the *RUNLENGTH*. After the first clock cycle the command controller enters into the *WAIT_FOR_RUNLENGTH* state until all the *RUNLENGTH* data has been consumed. Once finished and provided it is not the end of the line the command controller will return to the *PARSE* state.

vi   WAIT_FOR_NE

Similar to the *RUNLENGTH* commands the vertical commands can sometimes not find an edge in the current 16-bit frame. After the first clock cycle the command controller enters the *WAIT_FOR_NE* state and remains here until the edge is detected. Provided it is not the end of the line the command controller will return to the *PARSE* state.

vii   FINISH_LINE

At the end of a line the command controller needs to hold its data for the SFU before going back to the START state. Command controller remains in the *FINISH_LINE* state for one clock cycle to achieve this.

24.3.8   Next Edge Unit Sub-block Description

The *Next Edge Unit* (*NEU*) is responsible for detecting color changes, or edges, in the previous line based on the current address and color supplied by the Command Controller. The *NEU* is the interface to the SFU and it buffers the previous line for detecting an edge. For an edge detect

operation the Command Controller supplies the current address, this typically was the location of the last edge, but it could also be the end of a run length. With the current address a color is also supplied and using these two values the *NEU* will search the previous line for the next edge. If an edge is found the *NEU* returns this location to the Command Controller as the next address in the

5   current line and it sets a valid bit to tell the Command Controller that the edge has been detected. The Line Fill Unit uses this result to construct the current line. The *NEU* operates on 16-bit words and it is possible that there is no edge in the current 16 bits in the *NEU*. In this case the NEU will request more words from the SFU and will keep searching for an edge. It will continue doing this until it finds an edge or reaches the end of the previous line, which is based on the

10   *LINE_LENGTH*. A dataflow block diagram of the Next Edge unit is shown in Figure 152.

24.3.8.1   NEU Buffer

The algorithm being employed for decompression is based on the whole previous line and is not delineated during the line. However the *Next Edge Unit, NEU*, can only receive 16 bits at a time from the SFU. This presents a problem for vertical commands if the edge occurs in the successive

15   frame, but refers to a changing element in the current frame.

To accommodate this the *NEU* works on two frames at the same time, the current frame and the first 3 bits from the successive frame. This allows for the information that is needed from the previous line to construct the current frame of the current line.

In addition to this buffering there is also buffering right after the data is received from the SFU as

20   the SFU output is not registered. The current implementation of the SFU takes two clock cycles from when a request for a current line is received until it is returned and registered. However when NEU requests a new frame it needs it on the next clock cycle to maintain a decoded rate of 2 bits per clock cycle. A more detailed diagram of the buffer in the *NEU* is shown in Figure 153. The output of the buffer are two 16-bit vectors, *use_prev_line_a* and *use_prev_line_b*, that are

25   used to detect an edge that is relevant to the current line being put together in the Line Fill Unit.

24.3.8.2   NEU Edge Detect

The *NEU* Edge Detect block takes the two 16 bit vectors supplied by the buffer and based on the current line position in the current line, *a0*, and the current color, *sd_color*, it will detect if there is an edge relevant to the current frame. If the edge is found it supplies the current line position, *b1p*,

30   to the command controller and the line fill unit. The configuration of the edge detect is shown in Figure 154.

The two vectors from the buffer, *use_prev_line_a* and *use_prev_line_b*, pass into two sub-blocks, *transition_wtob* and *transition_btow*. *transition_wtob* detects if any white to black transitions occur

35   in the 19 bit vector supplied and outputs a 19-bit vector displaying the transitions. *transition_wtob* is functionally the same as *transition_btow*, but it detects white to black transitions.

The two 19-bit vectors produced enter into a multiplexer and the output of the multiplexer is controlled by *color_neu*. *color_neu* is the current edge transition color that the edge detect is searching for.

The output of the multiplexer is masked against a 19-bit vector, the mask is comprised of three parts concatenated together: *decode_b_ext*, *decode_b* and *FIRST_FLU_WRITE*.

The output of *transition_wtob* (and it complement *transition_btow*) are all the transitions in the 16 bit word that is under review. The *decode_b* is a mask generated from *a0*. In bit-wise terms all the bits above and including *a0* are 1's and all bits below *a0* are 0's. When they are gated together it means that all the transitions below *a0* are ignored and the first transition after *a0* is picked out as the next edge.

The *decode_b* block decodes the 4 lsb of the current address (*a0*) into 16-bit mask bits that control which of the data bits are examined. Table 158 shows the truth table for this block.

Table 158. Decode_b truth table

input	output
0000	1111111111111111
0001	1111111111111110
0010	1111111111111100
0011	1111111111111000
0100	1111111111110000
0101	1111111111100000
0110	1111111111000000
0111	1111111110000000
1000	1111111100000000
1001	1111111000000000
1010	1111110000000000
1011	1111100000000000
1100	1111000000000000
1101	1110000000000000
1110	1100000000000000
1111	1000000000000000

For cases when there is a negative vertical command from the stream decoder it is possible that the edge is in the three lower significant bits of the next frame. The *decode_b_ext* block supplies the mask so that the necessary bits can be used by the *NEU* to detect an edge if present, Table 159 shows the truth table for this block.

Table 159. Decode_b_ext truth table

delta	output
Vertical(-3)	111
Vertical(-2)	111
Vertical(-1)	011

375

OTHERS	001

FIRST_FLU_WRITE is only used in the first frame of the current line. 2.2.5 a) in [22] refers to "Processing the first picture element", in which it states that "The first starting picture element, *a0*, on each coding line is imaginarily set at a position *just* before the first picture element, and is

5   regarded as a white picture element". *transition_wtob* and *transition_btow* are set up produce this case for every single frame. However it is only used by the *NEU* if it is not masked out. This occurs when *FIRST_FLU_WRITE* is '1' which is only asserted at the beginning of a line.

2.2.5 b) in [22] covers the case of "Processing the last picture element", this case states that "The coding of the coding line continues until the position of the imaginary changing element situated

10   after the last actual element is coded". This means that no matter what the current color is the *NEU* needs to always find an edge at the end of a line. This feature is used with negative vertical commands.

The vector, *end_frame*, is a "one-hot" vector that is asserted during the last frame. It asserts a bit in the end of line position, as determined by *LineLength*, and this simulates an edge in this

15   location which is ORed with the transition's vector. The output of this, *masked_data*, is sent into the *encodeB_one_hot* block

24.3.8.3 Encode_b_one_hot

The *encode_b_one_hot* block is the first stage of a two stage process that encodes the data to determine the address of the 0 to 1 transition. Table 160 lists the truth table outlining the

20   functionally required by this block.

Table 160. Encode_b_one_hot Truth Table

Input	output
XXXXXXXXXXXXXXXXX1	0000000000000000001
XXXXXXXXXXXXXXXX10	0000000000000000010
XXXXXXXXXXXXXXX100	0000000000000000100
XXXXXXXXXXXXXX1000	0000000000000001000
XXXXXXXXXXXXX10000	0000000000000010000
XXXXXXXXXXXX100000	0000000000000100000
XXXXXXXXXXX1000000	0000000000001000000
XXXXXXXXXX10000000	0000000000010000000
XXXXXXXXX100000000	0000000000100000000
XXXXXXXX1000000000	0000000001000000000
XXXXXXX10000000000	0000000010000000000
XXXXXX100000000000	0000000100000000000
XXXXX1000000000000	0000001000000000000
XXXX10000000000000	0000010000000000000
XXXX100000000000000	0000100000000000000

XXX1000000000000000	0001000000000000000
XX10000000000000000	0010000000000000000
X100000000000000000	0100000000000000000
1000000000000000000	1000000000000000000
0000000000000000000	0000000000000000000

The output of *encode_b_one_hot* is a "one-hot" vector that will denote where that edge transition is located. In cases of multiple edges, only the first one will be picked.

24.3.8.4 Encode_b_4bit

Encode_b_4bit is the second stage of the two stage process that encodes the data to determine the address of the 0 to 1 transition.

Encode_b_4bit receives the "one-hot" vector from *encode_b_one_hot* and determines the bit location that is asserted. If there is none present this means that there was no edge present in this frame. If there is a bit asserted the bit location in the vector is converted to a number, for example if bit 0 is asserted then the number is one, if bit one is asserted then the number is one, etc. The delta supplied to the *NEU* determines what vertical command is being processed. The formula that is implemented to return *b1p* to the command controller is:

```
for V(n)  b1p = x + n modulus16
    where x is the number that was extracted from the "one-hot"
    vector and n is the vertical command.
```

24.3.8.5 State machine

The following is an explanation of all the states that the *NEU* state machine utilizes.

i   NEU_START

This is the state that NEU enters when a hard or soft reset occurs or when *Go* has been de-asserted. This state can not left until the reset has been removed, *Go* has been asserted and it detects that the command controller has entered it's *AWAIT_BUFF* state. When this occurs the *NEU* enters the *NEU_FILL_BUFF* state.

ii   NEU_FILL_BUFF

Before any compressed data can be decoded the *NEU* needs to fill up its buffer with new data from the SFU. The rest of the LBD waits while the *NEU* retrieves the first four frames from the previous line. Once completed it enters the *NEU_HOLD* state.

iii   NEU_HOLD

The NEU waits in this state for one clock cycle while data requested from the SFU on the last access returns.

iv   NEU_RUNNING

NEU_RUNNING controls the requesting of data from the SFU for the remainder of the line by pulsing *lbd_sfu_pladvword* when the LBD needs a new frame from the SFU. When the *NEU* has received all the word it needs for the current line, as denoted by the *LineLength*, the NEU enters the *NEU_EMPTY* state.

377

v   NEU_EMPTY

NEU waits in this state while the rest of the LBD finishes outputting the completed line to the SFU. The NEU leaves this state when *Go* gets deasserted. This occurs when the *end_of_line* signal is detected from the LBD.

24.3.9   Line Fill Unit sub-block description

The Line Fill Unit, LFU, is responsible for filling the next line buffer in the SFU. The SFU receives the data in blocks of sixteen bits. The LFU uses the color and *a0* provided by the Command Controller and when it has put together a complete 16-bit frame, it is written out to the SFU. The LBD signals to the SFU that the data is valid by strobing the *lbd_sfu_wdatavalid* signal.

When the LFU is at the end of the line for the current line data it strobes *lbd_sfu_advline* to indicate to the SFU that the end of the line has occurred.

A dataflow block diagram of the line fill unit is shown in Figure 154.

The dataflow above has the following blocks:

24.3.9.1 State Machine

The following is an explanation of all the states that the LFU state machine utilizes.

i   LFU_START

This is the state that the LFU enters when a hard or soft reset occurs or when *Go* has been de-asserted. This state can not left until the reset has been removed, *Go* has been asserted and it detects that *a0* is no longer zero, this only occurs once the command controller start processing data from the *Next Edge Unit, NEU*.

ii   LFU_NEW_REG

LFU_NEW_REG is only entered at the beginning of a new frame. It can remain in this state on subsequent cycles if a whole frame is completed in one clock cycle. If the frame is completed the LFU will output the data to the SFU with the write enable signal. However if a frame is not completed in one clock cycle the state machine will change to the *LFU_COMPLETE_REG* state to complete the remainder of the frame. *LFU_NEW_REG* handles all the *lbd_sfu_wdata* writes and asserts *lbd_sfu_wdatavalid* as necessary.

iii   LFU_COMPLETE_REG

LFU_COMPLETE_REG fills out all the remaining parts of the frame that were not completed in the first clock cycle. The command controller supplies the *a0* value and the color and the state machine uses these to derive the *limit* and *color_sel_16bit_lf* which the *line_fill_data* block needs to construct a frame. *Limit* is the four lower significant bits of *a0* and *color_sel_16bit_lf* is a 16-bit wide mask of *sd_color*. The state machine also maintains a check on the upper eleven bits of *a0*. If these increment from one clock cycle to the next that means that a frame is completed and the data can be written to the SFU. In the case of the *LineLength* being reached the Line Fill Unit fills out the remaining part of the frame with the color of the last bit in the line that was decoded.

24.3.9.2 line_fill_data

line_fill_data takes the *limit* value and the *color_sel_16bit_lf* values and constructs the current frame that the command controller and the next edge unit are decoding. The following pseudo

code illustrate the logic followed by the line_fill_data. *work_sfu_wdata* is exported by the LBD to the SFU as *lbd_sfu_wdata*.

```
if    (lfu_state    ==    LFU_START)    OR    (lfu_state    ==
    LFU_NEW_REG) then
        work_sfu_wdata = color_sel_16bit_lf
    else
        work_sfu_wdata[(15 - limit) downto limit] =
                color_sel_16bit_lf[(15 - limit) downto limit]
```

25   Spot FIFO Unit (SFU)

25.1   OVERVIEW

The Spot FIFO Unit (SFU) provides the means by which data is transferred between the LBD and the HCU. By abstracting the buffering mechanism and controls from both units, the interface is clean between the data user and the data generator. The amount of buffering can also be increased or decreased without affecting either the LBD or HCU. Scaling of data is performed in the horizontal and vertical directions by the SFU so that the output to the HCU matches the printer resolution. Non-integer scaling is supported in both the horizontal and vertical directions. Typically, the scale factor will be the same in both directions but may be programmed to be different.

25.2   MAIN FEATURES OF THE SFU

The SFU replaces the Spot Line Buffer Interface (SLBI) in PEC1. The spot line store is now located in DRAM.

The SFU outputs the previous line to the LBD, stores the next line produced by the LBD and outputs the HCU read line. Each interface to DRAM is via a feeder FIFO. The LBD interfaces to the SFU with a data width of 16 bits. The SFU interfaces to the HCU with a data width of 1 bit. Since the DRAM word width is 256-bits but the LBD line length is a multiple of 16 bits, a capability to flush the last multiples of 16-bits at the end of a line into a 256-bit DRAM word size is required. Therefore, SFU reads of DRAM words at the end of a line, which do not fill the DRAM word, will already be padded.

A signal *sfu_lbd_rdy* to the LBD indicates that the SFU is available for writing and reading. For the first LBD line after SFU *Go* has been asserted, previous line data is not supplied until after the first *lbd_sfu_advline* strobe from the LBD (zero data is supplied instead), and *sfu_lbd_rdy* to the LBD indicates that the SFU is available for writing. *lbd_sfu_advline* tells the SFU to advance to the next line. *lbd_sfu_pladvword* tells the SFU to supply the next 16-bits of previous line data. Until the number of *lbd_sfu_pladvword* strobes received is equivalent to the LBD line length, *sfu_lbd_rdy* indicates that the SFU is available for both reading and writing. Thereafter it indicates the SFU is available for writing. The LBD should not generate *lbd_sfu_pladvword* or *lbd_sfu_advline* strobes until *sfu_lbd_rdy* is asserted.

A signal *sfu_hcu_avail* indicates that the SFU has data to supply to the HCU. Another signal *hcu_sfu_advdot,* from the HCU, tells the SFU to supply the next dot. The HCU should not

generate the *hcu_sfu_advdot* signal until *sfu_hcu_avail* is true. The HCU can therefore stall waiting for the *sfu_hcu_avail* signal.

X and Y non-integer scaling of the bi-level dot data is performed in the SFU.

At 1600 dpi the SFU requires 1 dot per cycle for all DRAM channels, 3 dots per cycle in total (read + read + write). Therefore the SFU requires two 256 bit read DRAM access per 256 cycles, 1 write access every 256 cycles. A single DIU read interface will be shared for reading the current and previous lines from DRAM.

25.3   BI-LEVEL DRAM MEMORY BUFFER BETWEEN LBD, SFU AND HCU

Figure 158 shows a bi-level buffer store in DRAM. Figure 158 (a) shows the LBD previous line address reading after the HCU read line address in DRAM. Figure 158 (b) shows the LBD previous line address reading before the HCU read line address in DRAM.

Although the LBD and HCU read and write complete lines of data, the bi-level DRAM buffer is not line based. The buffering between the LBD, SFU and HCU is a FIFO of programmable size. The only line based concept is that the line the HCU is currently reading cannot be over-written because it may need to be re-read for scaling purposes.

The SFU interfaces to DRAM via three FIFOs:

a. The *HCUReadLineFIFO* which supplies dot data to the HCU.

b. The *LBDNextLineFIFO* which writes decompressed bi-level data from the LBD.

c. The *LBDPrevLineFIFO* which reads previous decompressed bi-level data for the LBD.

There are four address pointers used to manage the bi-level DRAM buffer:

a.   *hcu_readline_rd_adr[21:5]* is the read address in DRAM for the *HCUReadLineFIFO*.

b.   *hcu_startreadline_adr[21:5]* is the start address in DRAM for the current line being read by the *HCUReadLineFIFO*.

c.   *lbd_nextline_wr_adr[21:5]* is the write address in DRAM for the *LBDNextLineFIFO*.

d.   *lbd_prevline_rd_adr[21:5]* is the read address in DRAM for the *LBDPrevLineFIFO*.

The address pointers must obey certain rules which indicate whether they are valid:

a. *hcu_readline_rd_adr* is only valid if it is reading earlier in the line than *lbd_nextline_wr_adr* is writing i.e. the fifo is not empty

b. The SFU (*lbd_nextline_wr_adr*) cannot overwrite the current line that the HCU is reading from (*hcu_startreadline_adr*) i.e. the fifo is not full, when compared with the HCU read line pointer

c. The *LBDNextLineFIFO* (*lbd_nextline_wr_adr*) must be writing earlier in the line than *LBD-PrevLineFIFO* (*lbd_prevline_rd_adr*) is reading and must not overwrite the current line that the HCU is reading from i.e. the fifo is not full when compared to the *PrevLineFifo* read pointer

d. The *LBDPrevLineFIFO* (*lbd_prevline_rd_adr*) can read right up to the address that *LBDNext-LineFIFO* (*lbd_nextline_wr_adr*) is writing i.e the fifo is not empty.

e. At startup i.e. when *sfu_go* is asserted, the pointers are reset to *start_sfu_adr[21:5]*.

f. The address pointers can wrap around the SFU bi-level store area in DRAM.

As a guideline, the typical FIFO size should be a minimum of 2 lines stored in DRAM, nominally 3 lines, up to a programmable number of lines. A larger buffer allows lines to be decompressed in advance. This can be useful for absorbing local complexities in compressed bi-level images.

25.4 DRAM ACCESS REQUIREMENTS

The SFU has 1 read interface to the DIU and 1 write interface. The read interface is shared between the previous and current line read FIFOs.

The spot line store requires 5.1 Kbytes of DRAM to store 3 A4 lines. The SFU will read and write the spot line store in single 256-bit DRAM accesses. The SFU will need 256-bit double buffers for each of its previous, current and next line interfaces.

The SFU's DIU bandwidth requirements are summarized in Table 161.

Table 161. DRAM bandwidth requirements

Direction	Maximum number of cycles between each 256-bit DRAM access	Peak Bandwidth required to be supported by DIU (bits/cycle)	Average Bandwidth (bits/cycle)
Read	1281	2	2
Write	2562	1	1

1: Two separate reads of 1 bit/cycle.

2: Write at 1 bit/cycle.

25.5 SCALING

Scaling of bi-level data is performed in both the horizontal and vertical directions by the SFU so that the output to the HCU matches the printer resolution. The SFU supports non-integer scaling with the scale factor represented by a numerator and a denominator. Only scaling up of the bi-level data is allowed, i.e. the numerator should be greater than or equal to the denominator. Scaling is implemented using a counter as described in the pseudocode below. An advance pulse is generated to move to the next dot (x-scaling) or line (y-scaling).

```
if (count + denominator >= numerator) then
    count = (count + denominator) - numerator
    advance = 1
else
    count = count + denominator
    advance = 0
```

X scaling controls whether the SFU supplies the next dot or a copy of the current dot when the HCU asserts *hcu_sfu_advdot*. The SFU counts the number of *hcu_sfu_advdot* signals from the HCU. When the SFU has supplied an entire HCU line of data, the SFU will either re-read the current line from DRAM or advance to the next line of HCU read data depending on the programmed Y scale factor.

An example of scaling for *numerator* = 7 and *denominator* = 3 is given in Table 162. The signal *advance* if asserted causes the next input dot to be output on the next cycle, otherwise the same input dot is output

Table 162. Non-integer scaling example for scaleNum = 7, scaleDenom = 3

count	advance	dot
0	0	1
3	0	1
6	1	1
2	0	2
5	1	2
1	0	3
4	1	3
0	0	4
3	0	4
6	1	4
2	0	5

25.6   LEAD-IN AND LEAD-OUT CLIPPING

To account for the case where there may be two SoPEC devices, each generating its own portion of a dot-line, the first dot in a line may not be replicated the total scale-factor number of times by an individual SoPEC. The dot will ultimately be scaled-up correctly with both devices doing part of the scaling, one on its lead-out and the other on its lead in. Scaled up dots on the lead-out, i.e. which go beyond the HCU linelength, will be ignored. Scaling on the lead-in, i.e. of the first valid dot in the line, is controlled by setting the *XstartCount* register.

At the start of each line *count* in the pseudo-code above is set to *XstartCount*. If there is no lead-in, *XstartCount* is set to 0 i.e. the first value of *count* in Table . If there is lead-in then *XstartCount* needs to be set to the appropriate value of *count* in the sequence above.

25.7   INTERFACES BETWEEN LDB, SFU AND HCU

25.7.1   LDB-SFU Interfaces

The LBD has two interfaces to the SFU. The LBD writes the next line to the SFU and reads the previous line from the SFU.

25.7.1.1 LBDNextLineFIFO Interface

The *LBDNextLineFIFO* interface from the LBD to the SFU comprises the following signals:

- *lbd_sfu_wdata*, 16-bit write data.
- *lbd_sfu_wdatavalid*, write data valid.
- *lbd_sfu_advline*, signal indicating LDB has advanced to the next line.

The LBD should not write to the SFU until *sfu_lbd_rdy* is true. The LBD can therefore stall waiting for the *sfu_lbd_rdy* signal.

25.7.1.2 LBDPrevLineFIFO Interface

The *LBDPrevLineFIFO* interface from the SFU to the LBD comprises the following signals:

- *sfu_lbd_pldata*, 16-bit data.

The previous line read buffer interface from the LBD to the SDU comprises the following signals:

- *lbd_sfu_pladvword*, signal indicating to the SFU to supply the next 16-bit word.
- *lbd_sfu_advline,* signal indicating LDB has advanced to the next line.

Previous line data is not supplied until after the first *lbd_sfu_advline* strobe from the LBD (zero data is supplied instead). The LBD should not assert *lbd_sfu_pladvword* unless *sfu_lbd_rdy* is asserted.

25.7.1.3 Common Control Signals

sfu_lbd_rdy indicates to the LBD that the SFU is available for writing. After the first *lbd_sfu_advline* and before the number of *lbd_sfu_pladvword* strobes received is equivalent to the LBD line length, *sfu_lbd_rdy* indicates that the SFU is available for both reading and writing. Thereafter it indicates the SFU is available for writing.

The LBD should not generate *lbd_sfu_pladvword* or *lbd_sfu_advline* strobes until *sfu_lbd_rdy* is asserted.

25.7.2 SFU-HCU Current Line FIFO Interface

The interface from the SFU to the HCU comprises the following signals:

- *sfu_hcu_sdata,* 1-bit data.
- *sfu_hcu_avail,* data valid signal indicating that there is data available in the SFU *HCUReadLineFIFO.*

The interface from HCU to SFU comprises the following signals:

- *hcu_sfu_advdot*, indicating to the SFU to supply the next dot.

The HCU should not generate the *hcu_sfu_advdot* signal until *sfu_hcu_avail* is true. The HCU can therefore stall waiting for the *sfu_hcu_avail* signal.

25.8 IMPLEMENTATION

25.8.1 Definitions of IO

Table 163. SFU Port List

Port Name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	SoPEC Functional clock.
prst_n	1	In	Global reset signal.
DIU Read Interface signals			
sfu_diu_rreq	1	Out	SFU requests DRAM read. A read request must be accompanied by a valid read address.
sfu_diu_radr[21:5]	17	Out	Read address to DIU 17 bits wide (256-bit aligned word).
diu_sfu_rack	1	In	Acknowledge from DIU that read request has been accepted and new read address can be placed on *sfu_diu_radr.*
diu_data[63:0]	64	In	Data from DIU to SoPEC Units. First 64-bits are bits 63:0 of 256 bit word. Second 64-bits are bits 127:64 of 256 bit word. Third 64-bits are bits 191:128 of 256 bit word.

			Fourth 64-bits are bits 255:192 of 256 bit word.
diu_sfu_rvalid	1	In	Signal from DIU telling SoPEC Unit that valid read data is on the *diu_data* bus.

DIU Write Interface signals

sfu_diu_wreq	1	Out	SFU requests DRAM write. A write request must be accompanied by a valid write address together with valid write data and a write valid.
sfu_diu_wadr[21:5]	17	Out	Write address to DIU 17 bits wide (256-bit aligned word).
diu_sfu_wack	1	In	Acknowledge from DIU that write request has been accepted and new write address can be placed on *sfu_diu_wadr*.
sfu_diu_data[63:0]	64	Out	Data from SFU to DIU. First 64-bits are bits 63:0 of 256 bit word. Second 64-bits are bits 127:64 of 256 bit word. Third 64-bits are bits 191:128 of 256 bit word. Fourth 64-bits are bits 255:192 of 256 bit word.
sfu_diu_wvalid	1	Out	Signal from PEP Unit indicating that data on *sfu_diu_data* is valid.

PCU Interface data and control signals

pcu_adr[5:2]	4	In	PCU address bus. Only 4 bits are required to decode the address space for this block
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU
sfu_pcu_datain[31:0]	32	Out	Read data bus from the SFU to the PCU
pcu_rwn	1	In	Common read/not-write signal from the PCU
pcu_sfu_sel	1	In	Block select from the PCU. When *pcu_sfu_sel* is high both *pcu_adr* and *pcu_dataout* are valid
sfu_pcu_rdy	1	Out	Ready signal to the PCU. When *sfu_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *sfu_pcu_datain* is valid.

LBD Interface Data and Control Signals

sfu_lbd_rdy	1	Out	Signal indication that SFU has previous line data available and is ready to be written to.
lbd_sfu_advline	1	In	Line advance signal for both next and previous lines.
lbd_sfu_pladvword	1	In	Advance word signal for previous line buffer.
sfu_lbd_pldata[15:0]	16	Out	Data from the previous line buffer.

lbd_sfu_wdata[15:0]	16	In	Write data for next line buffer.
lbd_sfu_wdatavalid	1	In	Write data valid signal for next line buffer data.
HCU Interface Data and Control Signals			
hcu_sfu_advdot	1	In	Signal indicating to the SFU that the HCU is ready to accept the next dot of data from SFU.
sfu_hcu_sdata	1	Out	Bi-level dot data.
sfu_hcu_avail	1	Out	Signal indicating valid bi-level dot data on *sfu_hcu_sdata*.

25.8.2 Configuration Registers

Table 164. SFU Configuration Registers

Address (SFU_base +)	register name	#bits	value on reset	description
Control registers				
0x00	Reset	1	0x1	A write to this register causes a reset of the SFU. This register can be read to indicate the reset state: 0 - reset in progress 1 - reset not in progress
0x04	Go	1	0x0	Writing 1 to this register starts the SFU. Writing 0 to this register halts the SFU. When Go is deasserted the state-machines go to their idle states but all counters and configuration registers keep their values. When Go is asserted all counters are reset, but configuration registers keep their values (i.e. they don't get reset). The SFU must be started before the LBD is started. This register can be read to determine if the SFU is running (1 - running, 0 - stopped).
Setup registers (constant for during processing the page)				
0x08	HCUNumDots	16	0x0000	Width of HCU line (in dots).
0x0C	HCUDRAMW	8	0x00	Number of 256-bit DRAM words in a

	ords			HCU line - 1.
0x10	LBDDRAMWords	8	0x00	Number of 256-bit words in a LBD line - 1. (LBD line length must be at least 128 bits).
0x14	StartSfuAdr[21:5] (256-bit aligned DRAM address)	17	0x0000 0	First SFU location in memory.
0x18	EndSfuAdr[21:5] (256-bit aligned DRAM address)	17	0x0000 0	Last SFU location in memory.
0x1C	XstartCount	8	0x00	Value to be loaded at the start of every line into the counter used for scaling in the X direction. Used to control the scaling of the first dot in a line. This value will typically equal zero, except in the case where a number of dots are clipped on the lead in to a line. XstartCount must be programmed to be less than the XscaleNum value.
0x20	XscaleNum	8	0x01	Numerator of spot data scale factor in X direction.
0x24	XscaleDenom	8	0x01	Denominator of spot data scale factor in X direction.
0x28	YscaleNum	8	0x01	Numerator of spot data scale factor in Y direction.
0x2C	YscaleDenom	8	0x01	Denominator of spot data scale factor in Y direction.
Work registers (PCU has read-only access)				
0x30	HCUReadLineAdr[21:5]	17	-	Current address pointer in DRAM to HCU read data. Read only register.

	(256-bit aligned DRAM address)			
0x34	HCUStartReadLineAdr[21:5] (256-bit aligned DRAM address)	17	-	Start address in DRAM of line being read by HCU buffer in DRAM. Read only register.
0x38	LBDNextLineAdr[21:5] (256-bit aligned DRAM address)	17	-	Current address pointer in DRAM to LBD write data. Read only register
0x3C	LBDPrevLineAdr[21:5] (256-bit aligned DRAM address)	17	-	Current address pointer in DRAM to LBD read data. Read only register

25.8.3 SFU sub-block partition

The SFU contains a number of sub-blocks:

Name	description
PCU Interface	PCU interface, configuration and status registers. Also generates the *Go* and the *Reset* signals for the rest of the SFU
LBD Previous Line FIFO	Contains FIFO which is read by the LBD previous line interface.
LBD Next Line FIFO	Contains FIFO which is written by the LBD next line interface.
HCU Read Line FIFO	Contains FIFO which is read by the HCU interface.
DIU Interface and Address Generator	Contains DIU read interface and DIU write interface. Manages the address pointers for the bi-level DRAM buffer. Contains X and Y scaling logic.

5

The various FIFO sub-blocks have no knowledge of where in DRAM their read or write data is stored. In this sense the FIFO sub-blocks are completely de-coupled from the bi-level DRAM buffer. All DRAM address management is centralised in the DIU Interface and Address Generation sub-block. DRAM access is pre-emptive i.e. after a FIFO unit has made an access

5   then as soon as the FIFO has space to read or data to write a DIU access will be requested immediately. This ensures there are no unnecessary stalls introduced e.g. at the end of an LBD or HCU line.

There now follows a description of the SFU sub-blocks.

25.8.4   PCU Interface Sub-block

10   The PCU interface sub-block provides for the CPU to access SFU specific registers by reading or writing to the SFU address space.

25.8.5   LBDPrevLineFIFO sub-block

Table 165. LBDPrevLineFIFO Additional IO Definitions

Port Name	Pins	I/O	Description
Internal Output			
plf_rdy	1	Out	Signal indicating *LBDPrevLineFIFO* is ready to be read from. Until the first *lbd_sfu_advline* for a band has been received and after the number of reads from DRAM for a line is received is equal to *LBDDRAMWords*, *plf_rdy* is always asserted. During the second and subsequent lines *plf_rdy* is deasserted whenever the *LBDPrevLineFIFO* has one word left in the FIFO..
DIU and Address Generation sub-block Signals			
plf_diurreq	1	Out	Signal indicating the *LBDPrevLineFIFO* has 256-bits of data free.
plf_diurack	1	In	Acknowledge that read request has been accepted and *plf_diurreq* should be de-asserted.
plf_diurdata	1	In	Data from the DIU to *LBDPrevLineFIFO*. First 64-bits are bits 63:0 of 256 bit word. Second 64-bits are bits 127:64 of 256 bit word. Third 64-bits are bits 191:128 of 256 bit word. Fourth 64-bits is are bits 255:192 of 256 bit word.
plf_diurrvalid	1	In	Signal indicating data on *plf_diurdata* is valid.
plf_diuidle	1	Out	Signal indicating DIU state-machine is in the IDLE state.

15

25.8.5.1 General Description

The *LBDPrevLineFIFO* sub-block comprises a double 256-bit buffer between the LBD and the DIU Interface and Address Generator sub-block. The FIFO is implemented as 8 times 64-bit words. The FIFO is written by the DIU Interface and Address Generator sub-block and read by the

20   LBD.

Whenever 4 locations in the FIFO are free the FIFO will request 256-bits of data from the DIU Interface and Address Generation sub-block by asserting *plf_diurreq*. A signal *plf_diurack* indicates that the request has been accepted and *plf_diurreq* should be de-asserted.

The data is written to the FIFO as 64-bits on *plf_diurdata[63:0]* over 4 clock cycles. The signal
5   *plf_diurvalid* indicates that the data returned on *plf_diurdata[63:0]* is valid. *plf_diurvalid* is used to generate the FIFO write enable, *write_en*, and to increment the FIFO write address, *write_adr[2:0]*. If the *LBDPrevLineFIFO* still has 256-bits free then *plf_diurreq* should be asserted again.

The DIU Interface and Address Generation sub-block handles all address pointer management
10   and DIU interfacing and decides whether to acknowledge a request for data from the FIFO.

The state diagram of the *LBDPrevLineFIFO* DIU Interface is shown in Figure 163. If *sfu_go* is deasserted then the state-machine returns to its *idle* state.

The LBD reads 16-bit wide data from the *LBDPrevLineFIFO* on *sfu_lbd_pldata[15:0]*. *lbd_sfu_pladvword* from the LBD tells the *LBDPrevLineFIFO* to supply the next 16-bit word. The
15   FIFO control logic generates a signal *word_select* which selects the next 16-bits of the 64-bit FIFO word to output on *sfu_lbd_pldata[15:0]*. When the entire current 64-bit FIFO word has been read by the LBD *lbd_sfu_pladvword* will cause the next word to be popped from the FIFO.

Previous line data is not supplied until after the first *lbd_sfu_advline* strobe from the LBD after *sfu_go* is asserted (zero data is supplied instead). Until the first *lbd_sfu_advline* strobe after
20   *sfu_go lbd_sfu_pladvword* strobes are ignored.

The *LBDPrevLineFIFO* control logic uses a counter, *pl_count[7:0]*, to counts the number of *DRAM* read accesses for the line. When the *pl_count* counter is equal to the *LBDDRAMWords,* a complete line of data has been read by the LBD the *plf_rdy* is set high, and the counter is reset. It remains high until the next *lbd_sfu_advline* strobe from the LBD. On receipt of the *lbd_sfu_advline*
25   strobe the remaining data in the 256-bit word in the FIFO is ignored, and the FIFO *read_adr* is rounded up if required.

The *LBDPrevLineFIFO* generates a signal *plf_rdy* to indicate that it has data available. Until the first *lbd_sfu_advline* for a band has been received and after the number of DRAM reads for a line is equal to *LBDDRAMWords, plf_rdy* is always asserted. During the second and subsequent lines
30   *plf_rdy* is deasserted whenever the *LBDPrevLineFIFO* has one word left.

The last 256-bit word for a line read from DRAM can contain extra padding which should not be output to the LBD. This is because the number of 16-bit words per line may not fit exactly into a 256-bit DRAM word. When the count of the number of DRAM reads for a line is equal to *lbd_dram_words* the *LBDPrevLineFIFO* must adjust the FIFO write address to point to the next
35   256-bit word boundary in the FIFO for the next line of data. At the end of a line the read address must round up the nearest 256-bit word boundary and ignore the remaining 16-bit words. This can be achieved by considering the FIFO read address, *read_adr[2:0]*, will require 3 bits to address 8 locations of 64-bits. The next 256-bit aligned address is calculated by inverting the MSB of the *read_adr* and setting all other bits to 0.
40

```
if (read_adr[1:0] /= b00 AND lbd_sfu_advline == 1)then
    read_adr[1:0] = b00
    read_adr[2] = ~read_adr[2]
```

25.8.6   LBDNextLineFIFO sub-block

Table 166. LBDNextLineFIFO Additional IO Definition

Port Name	Pins	I/O	Description
LBDNextLineFIFO Interface Signals			
nlf_rdy	1	Out	Signal indicating *LBDNextLineFIFO* is ready to be written to i.e. there is space in the FIFO.
DIU and Address Generation sub-block Signals			
nlf_diuwreq	1	Out	Signal indicating the *LBDNextLineFIFO* has 256-bits of data for writing to the DIU.
nlf_diuwack	1	In	Acknowledge from DIU that write request has been accepted and write data can be output on *nlf_diuwdata* together with *nlf_diuwvalid*.
nlf_diuwdata	1	Out	Data from *LBDNextLineFIFO* to DIU Interface. First 64-bits is bits 63:0 of 256 bit word Second 64-bits is bits 127:64 of 256 bit word Third 64-bits is bits 191:128 of 256 bit word Fourth 64-bits is bits 255:192 of 256 bit word
nlf_diuwvalid	1	In	Signal indicating that data on *wlf_diuwdata* is valid.

25.8.6.1 General Description

The *LBDNextLineFIFO* sub-block comprises a double 256-bit buffer between the LBD and the DIU Interface and Address Generator sub-block. The FIFO is implemented as 8 times 64-bit words. The FIFO is written by the LBD and read by the DIU Interface and Address Generator. Whenever 4 locations in the FIFO are full the FIFO will request 256-bits of data to be written to the DIU Interface and Address Generator by asserting *nlf_diuwreq*. A signal *nlf_diuwack* indicates that the request has been accepted and *nlf_diuwreq* should be de-asserted. On receipt of *nlf_diuwack*, the data is sent to the DIU Interface as 64-bits on *nlf_diuwdata[63:0]* over 4 clock cycles. The signal *nlf_diuwvalid* indicates that the data on *nlf_diuwdata[63:0]* is valid. *nlf_diuwvalid* should be asserted with the smallest latency after *nlf_diuwack*. If the *LBDNextLineFIFO* still has 256-bits more to transfer then *nlf_diuwreq* should be asserted again. The state diagram of the *LBDNextLineFIFO* DIU Interface is shown in Figure 166. If *sfu_go* is deasserted then the state-machine returns to its *Idle* state.

The signal *nlf_rdy* indicates that the *LBDNextLineFIFO* has space for writing by the LBD. The LBD writes 16-bit wide data supplied on *lbd_sfu_wdata[15:0]*. *lbd_sfu_wvalid* indicates that the data is valid.

The *LBDNextLineFIFO* control logic counts the number of *lbd_sfu_wvalid* signals and is used to correctly address into the next line FIFO. The *lbd_sfu_wvalid* counter is rounded up to the nearest

256-bit word when a *lbd_sfu_advline* strobe is received from the LBD. Any data remaining in the FIFO is flushed to DRAM with padding being added to fill a complete 256-bit word.

25.8.7   sfu_lbd_rdy Generation

The signal *sfu_lbd_rdy* is generated by ANDing *plf_rdy* from the *LBDPrevLineFIFO* and *nlf_rdy*

5   from the *LBDNextLineFIFO*.

sfu_lbd_rdy indicates to the LBD that the SFU is available for writing i.e. there is space available in the *LBDNextLineFIFO*. After the first *lbd_sfu_advline* and before the number of *lbd_sfu_pladvword* strobes received is equivalent to the line length, *sfu_lbd_rdy* indicates that the SFU is available for both reading, i.e. there is data in the *LBDPrevLineFIFO*, and writing.

10   Thereafter it indicates the SFU is available for writing.

25.8.8   LBD-SFU Interfaces Timing Waveform Description

In Figure 167 and Figure 168, shows the timing of the data valid and ready signals between the SFU and LBD. A diagram and pseudocode is given for both read and write interfaces between the SFU and LBD.

15   *25.8.8.1 LBD-SFU write interface timing*

The main points to note from Figure 167 are:

- In clock cycle 1 *sfu_lbd_rdy* detects that it has only space to receive 2 more 16 bit words from the LBD after the current clock cycle.
- The data on *lbd_sfu_wdata* is valid and this is indicated by *lbd_sfu_wdatavalid* being
20   asserted.
- In clock cycle 2 *sfu_lbd_rdy* is deasserted however the LBD can not react to this signal until clock cycle 3. So in clock cycle 3 there is also valid data from the LBD which consumes the last available location available in the FIFO in the SFU (FIFO free level is zero).
- In clock cycle 4 and 5 the FIFO is read and 2 words become free in the FIFO.
25   - In cycle 4 the SFU determines that the FIFO has more room and asserts the ready signal on the next cycle.
- The LBD has entered a pause mode and waits for *sfu_lbd_rdy* to be asserted again, in cycle 5 the LBD sees the asserted ready signal and responds by writing one unit into the FIFO, in cycle 6.
30   - The SFU detects it has 2 spaces left in the FIFO and the current cycle is an active write (same as in cycle 1), and deasserts the ready on the next cycle.
- In cycle 7 the LBD did not have data to write into the FIFO, and so the FIFO remains with one space left
- The SFU toggles the ready signal every second cycle, this allows the LBD to write one unit
35   at a time to the FIFO.
- In cycle 9 the LBD responds to the single ready pulse by writing into the FIFO and consuming the last remaining unit free.

The write interface pseudocode for generating the ready is.

```
// ready generation pseudocode
40      if (fifo_free_level > 2)then
```

```
                  nlf_rdy = 1
              elsif (fifo_free_level == 2) then
                if (lbd_sfu_wdatavalid == 1)then
                  nlf_rdy = 0
                else
                  nlf_rdy = 1
              elsif (fifo_free_level == 1) then
                if (lbd_sfu_wdatavalid == 1)then
                  nlf_rdy = 0
                else
                  nlf_rdy = NOT(sfu_lbd_rdy)
              else
                nlf_rdy     = 0
              sfu_lbd_rdy = (nlf_rdy AND plf_rdy)
```

25.8.8.2 SFU-LBD read interface

The read interface is similar to the write interface except that read data (*sfu_lbd_pldata*) takes an extra cycle to respond to the data advance signal (*lbd_sfu_pladvword* signal).

It is not possible to read the FIFO totally empty during the processing of a line, one word must always remain in the FIFO. At the end of a line the fifo can be read to totally empty. This functionality is controlled by the SFU with the generation of the *plf_rdy* signal.

There is an apparent corner case on the read side which should be highlighted. On examination this turns out to not be an issue.

Scenario 1:

> *sfu_lbd_rdy* will go low when there is still is still 2 pieces of data in the FIFO. If there is a *lbd_sfu_pladvword* pulse in the next cycle the data will appear on *sfu_lbd_pldata[15:0]*.

Scenario 2:

> *sfu_lbd_rdy* will go low when there is still 2 pieces of data in the FIFO. If there is no *lbd_sfu_pladvword* pulse in the next cycle and it is not the end of the page then the SFU will read the data for the next line from DRAM and the read FIFO will fill more, *sfu_lbd_rdy* will assert again, and so the data will appear on *sfu_lbd_pldata[15:0]*. If it happens that the next line of data is not available yet the *sfu_lbd_pldata* bus will go invalid until the next lines data is available. The LBD does not sample the *sfu_lbd_pldata* bus at this time (i.e. after the end of a line) and it is safe to have invalid data on the bus.

Scenario 3:

> *sfu_lbd_rdy* will go low when there is still 2 pieces of data in the FIFO. If there is no *lbd_sfu_pladvword* pulse in the next cycle and it is the end of the page then the SFU will do no more reads from DRAM, *sfu_lbd_rdy* will remain de-asserted, and the data will not be read out from the FIFO. However last line of data on the page is not needed for decoding in the LBD and will not be read by the LBD. So scenario 3 will never apply.

The pseudocode for the read FIFO ready generation

```
// ready generation pseudocode
```

```
if (pl_count == lbd_dram_words) then
   plf_rdy = 1
elsif (fifo_fill_level > 3)then
   plf_rdy = 1
elsif (fifo_fill_level == 3) then
   if (lbd_sfu_pladvword == 1)then
      plf_rdy = 0
   else
      plf_rdy = 1
elsif (fifo_fill_level == 2) then
   if (lbd_sfu_pladvword == 1)then
      plf_rdy = 0
   else
      plf_rdy = NOT(sfu_lbd_rdy)
else
   plf_rdy     = 0
sfu_lbd_rdy = (plf_rdy AND nlf_rdy)
```

25.8.9   HCUReadLineFIFO sub-block

Table 167. HCUReadLineFIFO Additional IO Definition

Port Name	Pins	I/O	Description
DIU and Address Generation sub-block Signals			
hrf_xadvance	1	In	Signal from horizontal scaling unit 1 - supply the next dot 1 - supply the current dot
hrf_hcu_endofline	1	Out	Signal lasting 1 cycle indicating then end of the HCU read line.
hrf_diurreq	1	Out	Signal indicating the *HCUReadLineFIFO* has space for 256-bits of DIU data.
hrf_diurack	1	In	Acknowledge that read request has been accepted and *hrf_diurreq* should be de-asserted.
hrf_diurdata	1	In	Data from *HCUReadLineFIFO* to DIU. First 64-bits are bits 63:0 of 256 bit word. Second 64-bits are bits 127:64 of 256 bit word. Third 64-bits are bits 191:128 of 256 bit word. Fourth 64-bits are bits 255:192 of 256 bit word.
hrf_diurvalid	1	In	Signal indicating data on *hrf_diurdata* is valid.
hrf_diuidle	1	Out	Signal indicating DIU state-machine is in the IDLE state.

25.8.9.1 General Description

The *HCUReadLineFIFO* sub-block comprises a double 256-bit buffer between the HCU and the DIU Interface and Address Generator sub-block. The FIFO is implemented as 8 times 64-bit words. The FIFO is written by the DIU Interface and Address Generator sub-block and read by the HCU.

5   The DIU Interface and Address Generation (DAG) sub-block interface of the *HCUReadLineFIFO* is identical to the *LBDPrevLineFIFO* DIU interface.

Whenever 4 locations in the FIFO are free the FIFO will request 256-bits of data from the DAG sub-block by asserting *hrf_diurreq*. A signal *hrf_diurack* indicates that the request has been accepted and *hrf_diurreq* should be de-asserted.

10   The data is written to the FIFO as 64-bits on *hrf_diurdata[63:0]* over 4 clock cycles. The signal *hrf_diurvalid* indicates that the data returned on *hrf_diurdata[63:0]* is valid. *hrf_diurvalid* is used to generate the FIFO write enable, *write_en*, and to increment the FIFO write address, *write_adr[2:0]*. If the *HCUReadLineFIFO* still has 256-bits free then *hrf_diurreq* should be asserted again.

15   The *HCUReadLineFIFO* generates a signal *sfu_hcu_avail* to indicate that it has data available for the HCU. The HCU reads single-bit data supplied on *sfu_hcu_sdata*. The FIFO control logic generates a signal *bit_select* which selects the next bit of the 64-bit FIFO word to output on *sfu_hcu_sdata*. The signal *hcu_sfu_advdot* tells the *HCUReadLineFIFO* to supply the next dot (*hrf_xadvance* = 1) or the current dot (*hrf_xadvance* = 0) on *sfu_hcu_sdata* according to the

20   *hrf_xadvance* signal from the scaling control unit in the DAG sub-block. The HCU should not generate the *hcu_sfu_advdot* signal until *sfu_hcu_avail* is true. The HCU can therefore stall waiting for the *sfu_hcu_avail* signal.

When the entire current 64-bit FIFO word has been read by the HCU *hcu_sfu_advdot* will cause the next word to be popped from the FIFO.

25   The last 256-bit word for a line read from DRAM and written into the *HCUReadLineFIFO* can contain dots or extra padding which should not be output to the HCU. A counter in the *HCUReadLineFIFO, hcuadvdot_count[15:0],* counts the number of *hcu_sfu_advdot* strobes received from the HCU. When the count equals *hcu_num_dots[15:0]* the *HCUReadLineFIFO* must adjust the FIFO read address to point to the next 256-bit word boundary in the FIFO. This

30   can be achieved by considering the FIFO read address, *read_adr[2:0]*, will require 3 bits to address 8 locations of 64-bits. The next 256-bit aligned address is calculated by inverting the MSB of the read_adr and setting all other bits to 0.

```
        If (hcuadvdot_count == hcu_num_dots) then
35          read_adr[1:0] = b00
            read_adr[2] = ~read_adr[2]
```

The DIU Interface and Address Generator sub-block scaling unit also needs to know when *hcuadvdot_count* equals *hcu_num_dots*. This condition is exported from the *HCUReadLineFIFO*

40   as the signal *hrf_hcu_endofline*. When the *hrf_hcu_endofline* is asserted the scaling unit will

decide based on vertical scaling whether to go back to the start of the current line or go onto the next line.

25.8.9.2 DRAM Access Limitation

The SFU must output 1 bit/cycle to the HCU. Since *HCUNumDots* may not be a multiple of 256 bits the last 256-bit DRAM word on the line can contain extra zeros. In this case, the SFU may not be able to provide 1 bit/cycle to the HCU. This could lead to a stall by the SFU. This stall could then propagate if the margins being used by the HCU are not sufficient to hide it. The maximum stall can be estimated by the calculation: DRAM service period - X scale factor * dots used from last DRAM read for HCU line.

25.8.10   DIU Interface and Address Generator Sub-block

Table 168. DIU Interface and Address Generator Additional IO Description

Port name	Pins	I/O	Description
Internal LBDPrevLineFIFO Inputs			
plf_diurreq	1	In	Signal indicating the *LBDPrevLineFIFO* has 256-bits of data free.
plf_diurack	1	Out	Acknowledge that read request has been accepted and *plf_diurreq* should be de-asserted.
plf_diurdata	1	Out	Data from the DIU to *LBDPrevLineFIFO*. First 64-bits are bits 63:0 of 256 bit word Second 64-bits are bits 127:64 of 256 bit word Third 64-bits are bits 191:128 of 256 bit word Fourth 64-bits are bits 255:192 of 256 bit word
plf_diurrvalid	1	Out	Signal indicating data on *plf_diurdata* is valid.
plf_diuidle	1	In	Signal indicating DIU state-machine is in the IDLE state.
Internal LBDNextLineFIFO Inputs			
nlf_diuwreq	1	In	Signal indicating the *LBDNextLineFIFO* has 256-bits of data for writing to the DIU.
nlf_diuwack	1	Out	Acknowledge from DIU that write request has been accepted and write data can be output on *nlf_diuwdata* together with *nlf_diuwvalid*.
nlf_diuwdata	1	In	Data from *LBDNextLineFIFO* to DIU Interface. First 64-bits are bits 63:0 of 256 bit word Second 64-bits are bits 127:64 of 256 bit word Third 64-bits are bits 191:128 of 256 bit word Fourth 64-bits are bits 255:192 of 256 bit word
nlf_diuwvalid	1	In	Signal indicating that data on *wlf_diuwdata* is valid.

Internal HCUReadLineFIFO Inputs			
hrf_hcu_endofline	1	In	Signal lasting 1 cycle indicating then end of the HCU read line.
hrf_xadvance	1	Out	Signal from horizontal scaling unit 1 - supply the next dot 1 - supply the current dot
hrf_diurreq	1	In	Signal indicating the *HCUReadLineFIFO* has space for 256-bits of DIU data.
hrf_diurack	1	Out	Acknowledge that read request has been accepted and *hrf_diurreq* should be de-asserted.
hrf_diurdata	1	Out	Data from *HCUReadLineFIFO* to DIU. First 64-bits are bits 63:0 of 256 bit word Second 64-bits are bits 127:64 of 256 bit word Third 64-bits are bits 191:128 of 256 bit word Fourth 64-bits are bits 255:192 of 256 bit word
hrf_diurvalid	1	Out	Signal indicating data on *plf_diurdata* is valid.
hrf_diuidle	1	In	Signal indicating DIU state-machine is in the IDLE state.

25.8.10.1 General Description

The DIU Interface and Address Generator (*DAG*) sub-block manages the bi-level buffer in DRAM. It has a DIU Write Interface for the *LBDNextLineFIFO* and a DIU Read Interface shared between the *HCUReadLineFIFO* and *LBDPrevLineFIFO*.

5   All DRAM address management is centralised in the *DAG*. DRAM access is pre-emptive i.e. after a FIFO unit has made an access then as soon as the FIFO has space to read or data to write a DIU access will be requested immediately. This ensures there are no unnecessary stalls introduced e.g. at the end of an LBD or HCU line.

The control logic for horizontal and vertical non-integer scaling logic is completely contained in the
10   *DAG* sub-block. The scaling control unit exports the *hlf_xadvance* signal to the *HCUReadLineFIFO* which indicates whether to replicate the current dot or supply the next dot for horizontal scaling.

25.8.10.2 DIU Write Interface

The *LBDNextLineFIFO* generates all the DIU write interface signals directly except for
15   *sfu_diu_wadr[21:5]* which is generated by the Address Generation logic

The DIU request from the *LBDNextLineFIFO* will be negated if its respective address pointer in DRAM is invalid i.e. *nlf_adrvalid* = 0. The implementation must ensure that no erroneous requests occur on *sfu_diu_wreq*.

25.8.10.3 DIU Read Interface

20   Both *HCUReadLineFIFO* and *LBDPrevLineFIFO* share the read interface. If both sources request simultaneously, then the arbitration logic implements a round-robin sharing of read accesses between the *HCUReadLineFIFO* and *LBDPrevLineFIFO*.

The DIU read request arbitration logic generates a signal, *select_hrfplf*, which indicates whether the DIU access is from the *HCUReadLineFIFO* or *LBDPrevLineFIFO* (0=*HCUReadLineFIFO*, 1 = *LBDPrevLineFIFO*). Figure 171 shows *select_hrfplf* multiplexing the returned DIU acknowledge and read data to either the *HCUReadLineFIFO* or *LBDPrevLineFIFO*.

5   The DIU read request arbitration logic is shown in Figure 172. The arbitration logic will select a DIU read request on *hrf_diurreq* or *plf_diurreq* and assert *sfu_diu_rreq* which goes to the DIU. The accompanying DIU read address is generated by the Address Generation Logic. The select signal *select_hrfplf* will be set according to the arbitration winner (0=*HCUReadLineFIFO*, 1=*LBDPrevLineFIFO*). *sfu_diu_rreq* is cleared when the DIU acknowledges the request on
10   *diu_sfu_rack*. Arbitration cannot take place again until the DIU state-machine of the arbitration winner is in the idle state, indicated by *diu_idle*. This is necessary to ensure that the DIU read data is multiplexed back to the FIFO that requested it.

The DIU read requests from the *HCUReadLineFIFO* and *LBDPrevLineFIFO* will be negated if their respective addresses in DRAM are invalid, *hrf_adrvalid* = 0 or *plf_adrvalid* = 0. The
15   implementation must ensure that no erroneous requests occur on *sfu_diu_rreq*.

If the *HCUReadLineFIFO* and *LBDPrevLineFIFO* request simultaneously, then if the request is not following immediately another DIU read port access, the arbitration logic will choose the *HCUReadLineFIFO* by default. If there are back to back requests to the DIU read port then the arbitration logic implements a round-robin sharing of read accesses between the
20   *HCUReadLineFIFO* and *LBDPrevLineFIFO*.

A pseudo-code description of the DIU read arbitration is given below.

```
       // history is of type {none, hrf, plf}, hrf is
       HCUReadLineFIFO, plf is LBDPrevLineFIFO
25         // initialisation on reset
       select_hrfplf = 0 // default choose hrf
       history = none // no DIU read access immediately preceding

           // state-machine is busy between asserting sfu_diu_rreq
30     and diu_idle = 1
           // if DIU read requester state-machine is in idle state
       then de-assert busy
           if (diu_idle == 1) then
             busy = 0
35
           //if acknowledge received from DIU then de-assert DIU
       request
           if (diu_sfu_rack == 1) then
             //de-assert request in response to acknowledge
40             sfu_diu_rreq = 0

           // if not busy then arbitrate between incoming requests
           // if request detected then assert busy
```

```
                        if (busy == 0) then
                            //if there is no request
                            if (hrf_diurreq == 0) AND (plf_diurreq == 0) then
                                sfu_diu_rreq = 0
                                history = none
                            // else there is a request
                            else {
                                // assert busy and request DIU read access
                                busy = 1
                                sfu_diu_rreq = 1
                                // arbitrate in round-robin fashion between the
requestors
                                // if only HCUReadLineFIFO requesting choose
HCUReadLineFIFO
                                if (hrf_diurreq == 1) AND (plf_diurreq == 0) then
                                    history = hrf
                                    select_hrfplf = 0
                                // if only LBDPrevLineFIFO requesting choose
LBDPrevLineFIFO
                                if (hrf_diurreq == 0) AND (plf_diurreq == 1) then
                                    history = plf
                                    select_hrfplf = 1
                                //if both HCUReadLineFIFO and LBDPrevLineFIFO
requesting
                                if (hrf_diurreq == 1) AND (plf_diurreq == 1) then
                                    // no immediately preceding request choose
HCUReadLineFIFO
                                    if (history == none) then
                                        history = hrf
                                        select_hrfplf = 0
                                    // if previous winner was HCUReadLineFIFO choose
LBDPrevLineFIFO
                                    elsif (history == hrf) then
                                        history = plf
                                        select_hrfplf = 1
                                    // if previous winner was LBDPrevLineFIFO choose
HCUReadLineFIFO
                                    elsif (history == plf) then
                                        history = hrf
                                        select_hrfplf = 0
                            // end there is a request
                            }
```

25.8.10.4   Address Generation Logic

The DIU interface generates the DRAM addresses of data read and written by the SFU's FIFOs.

A write request from the *LBDNextLineFIFO* on *nlf_diuwreq* causes a write request from the DIU Write Interface. The Address Generator supplies the DRAM write address on *sfu_diu_wadr[21:5]*.

A winning read request from the DIU read request arbitration logic causes a read request from the DIU Read Interface. The Address Generator supplies the DRAM read address on *sfu_diu_radr[21:5]*.

The address generator is configured with the number of DRAM words to read in a HCU line, *hcu_dram_words*, the first DRAM address of the SFU area, *start_sfu_adr[21:5]*, and the last DRAM address of the SFU area, *end_sfu_adr[21:5]*.

Note *hcu_dram_words* configuration register specifies the the number of DRAM words consumed per line in the HCU, while *lbd_dram_words* specifies the number of DRAM words generated per line by the LBD. These values are not required to be the same.

For example the LBD may store 10 DRAM words per line (*lbd_dram_words* = 10), but the HCU may consume 5 DRAM words per line. In such case the *hcu_dram_words* would be set to 5 and the HCU Read Line FIFO would trigger a new line after it had consumed 5 DRAM words (via *hrf_hcu_endofline*).

Address Generation

There are four address pointers used to manage the bi-level DRAM buffer:

a. *hcu_readline_rd_adr* is the read address in DRAM for the *HCUReadLineFIFO*.

b. *hcu_startreadline_adr* is the start address in DRAM for the current line being read by the *HCUReadLineFIFO*.

c. *lbd_nextline_wr_adr* is the write address in DRAM for the *LBDNextLineFIFO*.

d. *lbd_prevline_rd_adr* is the read address in DRAM for the *LBDPrevLineFIFO*.

The current value of these address pointers are readable by the CPU.

Four corresponding address valid flags are required to indicate whether the address pointers are valid, based on whether the FIFOs are full or empty.

a. *hlf_adrvalid*, derived from *hrf_nlf_fifo_emp*

b. *hlf_start_adrvalid*, derived from *start_hrf_nlf_fifo_emp*

c. *nlf_adrvalid*, derived from *nlf_plf_fifo_full* and *nlf_hrf_fifo_full*

d. *plf_adrvalid*, derived from *plf_nlf_fifo_emp*

DRAM requests from the FIFOs will not be issued to the DIU until the appropriate address flag is valid.

Once a request has been acknowledged, the address generation logic can calculate the address of the next 256-bit word in DRAM, ready for the next request.

Rules for address pointers

The address pointers must obey certain rules which indicate whether they are valid:

a. *hcu_readline_rd_adr* is only valid if it is reading earlier in the line than *lbd_nextline_wr_adr* is writing i.e. the fifo is not empty

b. The SFU (*lbd_nextline_wr_adr*) cannot overwrite the current line that the HCU is reading from (*hcu_startreadline_adr*) i.e. the fifo is not full, when compared with the HCU read line pointer

c. The *LBDNextLineFIFO* (*lbd_nextline_wr_adr*) must be writing earlier in the line than *LBD-PrevLineFIFO* (*lbd_prevline_rd_adr*) is reading and must not overwrite the current line that the HCU is reading from i.e. the fifo is not full when compared to the *PrevLineFifo* read pointer

d. The *LBDPrevLineFIFO* (*lbd_prevline_rd_adr*) can read right up to the address that *LBDNext-LineFIFO* (*lbd_nextline_wr_adr*) is writing i.e the fifo is not empty.

e. At startup i.e. when *sfu_go* is asserted, the pointers are reset to *start_sfu_adr[21:5]*.

f. The address pointers can wrap around the SFU bi-level store area in DRAM.

5   Address generator pseudo-code:

Initialization:

```
if (sfu_go rising edge) then
   //initialise address pointers to start of SFU address
   space
      lbd_prevline_rd_adr          = start_sfu_adr[21:5]
      lbd_nextline_wr_adr          = start_sfu_adr[21:5]
      hcu_readline_rd_adr          = start_sfu_adr[21:5]
      hcu_startreadline_adr        = start_sfu_adr[21:5]
      lbd_nextline_wr_wrap         = 0
      lbd_prevline_rd_wrap         = 0
      hcu_startreadline_wrap       = 0
      hcu_readline_rd_wrap         = 0
   }
```

Determine FIFO fill and empty status:

```
// calculate which FIFOs are full and empty
plf_nlf_fifo_emp    =    (lbd_prevline_rd_adr         ==
lbd_nextline_wr_adr) AND
                         (lbd_prevline_rd_wrap        ==
lbd_nextline_wr_wrap)
nlf_plf_fifo_full   =    (lbd_nextline_wr_adr         ==
lbd_prevline_rd_adr) AND
                         (lbd_prevline_rd_wrap        !=
lbd_nextline_wr_wrap)
nlf_hrf_fifo_full   =    (lbd_nextline_wr_adr         ==
hcu_startreadline_adr ) AND
                         (hcu_startreadline_wrap      !=
lbd_nextline_wr_wrap )
// hcu start address can jump addresses and so needs
comparitor
if (hcu_startreadline_wrap == lbd_nextline_wr_wrap) then
   start_hrf_nlf_fifo_emp   =            (hcu_startreadline_adr
>=lbd_nextline_wr_adr)
else
   start_hrf_nlf_fifo_emp       =        NOT(hcu_startreadline_adr
>=lbd_nextline_wr_adr)
// hcu read address can jump addresses and so needs
comparitor
if (hcu_readline_rd_wrap == lbd_nextline_wr_wrap) then
   hrf_nlf_fifo_emp    =                    (hcu_readline_rd_adr
>=lbd_nextline_wr_adr)
```

```
else
   hrf_nlf_fifo_emp          =          NOT(hcu_readline_rd_adr
>=lbd_nextline_wr_adr)
```

Address pointer updating:

```
// LBD Next line FIFO
// if DIU write acknowledge and LBDNextLineFIFO is not full
with reference to PLF and HRF
if (diu_sfu_wack  ==  1  AND  nlf_plf_fifo_full  !=  1  AND
nlf_hrf_fifo_full !=1 ) then
   if   (lbd_nextline_wr_adr   ==   end sfu adr)   then
// if end of SFU address range
      lbd_nextline_wr_adr = start_sfu_adr               //
go to start of SFU address range
      lbd_nextline_wr_wrap= NOT (lbd_nextline_wr_wrap)  //
invert the wrap bit
   else
      lbd_nextline_wr_adr++                              //
increment address pointer


// LBD PrevLine FIFO
//if DIU read acknowledge and LBDPrevLineFIFO is not empty
if  (diu_sfu_rack  ==  1  AND  select_hrfplf  ==  1  AND
plf_nlf_fifo_emp !=1) then
   if (lbd_prevline_rd_adr == end_sfu_adr) then
      lbd_prevline_rd_adr = start_sfu_adr               //
go to start of SFU address range
      lbd_prevline_rd_wrap= NOT (lbd_prevline_rd_wrap)  //
invert the wrap bit
   else
      lbd_prevline_rd_adr++                             //
increment address pointer


// HCU ReadLine FIFO
// if DIU read acknowledge and HCUReadLineFIFO fifo is not
empty
if  (diu_sfu_rack  ==  1  AND  select_hrfplf  ==  0  AND
hrf_nlf_fifo_emp != 1) then
   // going to update hcu read line address
   if (hrf_hcu_endofline == 1) AND (hrf_yadvance == 1) then {
// read the next line from DRAM
      // advance to start of next HCU line in DRAM
      hcu_startreadline_adr    =    hcu_startreadline_adr    +
lbd_dram_words
      offset  =  hcu_startreadline_adr  -  end_sfu_adr  -  1
// allow for address wraparound
```

```
                  if (offset >= 0) then
                    hcu_startreadline_adr = start_sfu_adr + offset
                          hcu_startreadline_wrap=
        NOT(hcu_startreadline_wrap)
                    hcu_readline_rd_adr = hcu_startreadline_adr
                    hcu_readline_rd_wrap= hcu_startreadline_wrap
                    }
                  elsif (hrf_hcu_endofline == 1) AND (hrf_yadvance == 0)
                  then
                    hcu_readline_rd_adr = hcu_startreadline_adr          //
        restart and re-use the same line
                    hcu_readline_rd_wrap= hcu_startreadline_wrap
                  elsif    (hcu readline rd adr    ==    end sfu adr)    then
                  // check if the FIFO needs to wrap space
                    hcu_readline_rd_adr = start_sfu_adr                  //
        go to start of SFU address space
                    hcu_readline_rd_wrap= NOT (hcu_readline_rd_wrap)
                  else
                    hcu_readline_rd_adr ++                               //
        increment address pointer
```

25.8.10.4.1   X scaling of data for HCUReadLineFIFO

The signal *hcu_sfu_advdot* tells the *HCUReadLineFIFO* to supply the next dot or the current dot on *sfu_hcu_sdata* according to the *hrf_xadvance* signal from the scaling control unit. When *hrf_xadvance* is 1 the *HCUReadLineFIFO* should supply the next dot. When *hrf_xadvance* is 0 the *HCUReadLineFIFO* should supply the current dot.

The algorithm for non-integer scaling is described in the pseudocode below. Note, *x_scale_count* should be loaded with *x_start_count* after reset and at the end of each line. The end of the line is indicated by *hrf_hcu_endofline* from the *HCUReadLineFIFO*.

```
                  if (hcu_sfu_advdot == 1) then
                    if (x_scale_count + x_scale_denom - x_scale_num >= 0)
                  then
                        x_scale_count  =  x_scale_count  +  x_scale_denom  -
        x_scale_num
                        hrf_xadvance = 1
                    else
                        x_scale_count = x_scale_count + x_scale_denom
                        hrf_xadvance = 0
                  else
                    x_scale_count = x_scale_count
                    hrf_xadvance = 0
```

25.8.10.4.2   Y scaling of data for HCUReadLineFIFO

The *HCUReadLineFIFO* counts the number of *hcu_sfu_advdot* strobes received from the HCU. When the count equals *hcu_num_dots* the *HCUReadLineFIFO* will assert *hrf_hcu_endofline* for a cycle.

The algorithm for non-integer scaling is described in the pseudocode below. Note, *y_scale_count* should be loaded with zero after reset.

```
if (hrf_hcu_endofline == 1) then
    if (y_scale_count + y_scale_denom - y_scale_num >= 0)
then
        y_scale_count = y_scale_count + y_scale_denom -
y_scale_num
        hrf_yadvance = 1
    else
        y_scale_count = y_scale_count + y_scale_denom
        hrf_yadvance = 0
else
    y_scale_count = y_scale_count
    hrf_yadvance = 0
```

When the *hrf_hcu_endofline* is asserted the Y scaling unit will decide whether to go back to the start of the current line, by setting *hrf_yadvance* = 0, or go onto the next line, by setting *hrf_yadvance* = 1.

Figure 176 shows an overview of X and Y scaling for HCU data.

26   Tag Encoder (TE)

26.1   OVERVIEW

The Tag Encoder (TE) provides functionality for Netpage-enabled applications, and typically requires the presence of IR ink (although K ink can be used for tags in limited circumstances). The TE encodes fixed data for the page being printed, together with specific tag data values into an error-correctable encoded tag which is subsequently printed in infrared or black ink on the page. The TE places tags on a triangular grid, and can be programmed for both landscape and portrait orientations.

Basic tag structures are normally rendered at 1600 dpi, while tag data is encoded into an arbitrary number of printed dots. The TE supports integer scaling in the Y-direction while the TFU supports integer scaling in the X-direction. Thus, the TE can render tags at resolutions less than 1600 dpi which can be subsequently scaled up to 1600 dpi.

The output from the TE is buffered in the Tag FIFO Unit (TFU) which is in turn used as input by the HCU. In addition, a *te_finishedband* signal is output to the end of band unit once the input tag data has been loaded from DRAM. The high level data path is shown by the block diagram in Figure 177.

After passing through the HCU, the tag plane is subsequently printed with an infrared-absorptive ink that can be read by a Netpage sensing device. Since black ink can be IR absorptive, limited functionality can be provided on offset-printed pages using black ink on otherwise blank areas of

403

the page - for example to encode buttons. Alternatively an invisible infrared ink can be used to print the position tags over the top of a regular page. However, if invisible IR ink is used, care must be taken to ensure that any other printed information on the page is printed in infrared-*transparent* CMY ink, as black ink will obscure the infrared tags. The monochromatic scheme was chosen to maximize dynamic range in blurry reading environments.

When multiple SoPEC chips are used for printing the same side of a page, it is possible that a single tag will be produced by two SoPEC chips. This implies that the TE must be able to print partial tags.

The throughput requirement for the SoPEC TE is to produce tags at half the rate of the PEC1 TE. Since the TE is reused from PEC1, the SoPEC TE over-produces by a factor of 2.

In PEC1, in order to keep up with the HCU which processes 2 dots per cycle, the tag data interface has been designed to be capable of encoding a tag in 63 cycles. This is actually accomplished in approximately 52 cycles within PEC1. If the SoPEC TE were to be modified from two dots production per cycle to a nominal one dot per cycle it should not lose the 63/52 cycle performance edge attained in the PEC1 TE.

26.2   WHAT ARE TAGS?

The first barcode was described in the late 1940's by Woodland and Silver, and finally patented in 1952 (US Patent 2,612,994) when electronic parts were scarce and very expensive. Now however, with the advent of cheap and readily available computer technology, nearly every item purchased from a shop contains a barcode of some description on the packaging. From books to CDs, to grocery items, the barcode provides a convenient way of identifying an object by a product number. The exact interpretation of the product number depends on the type of barcode. Warehouse inventory tracking systems let users define their own product number ranges, while inventory in shops must be more universally encoded so that products from one company don't overlap with products from another company. Universal Product Codes (UPC) were introduced in the mid 1970's at the request of the National Association of Food Chains for this very reason.

Barcodes themselves have been specified in a large number of formats. The older barcode formats contain characters that are displayed in the form of lines. The combination of black and white lines describe the information the barcodes contains. Often there are two types of lines to form the complete barcode: the characters (the information itself) and lines to separate blocks for better optical recognition. While the information may change from barcode to barcode, the lines to separate blocks stays constant. The lines to separate blocks can therefore be thought of as part of the constant structural components of the barcode.

Barcodes are read with specialized reading devices that then pass the extracted data onto the computer for further processing. For example, a point-of-sale scanning device allows the sales assistant to add the scanned item to the current sale, places the name of the item and the price on a display device for verification etc. Light-pens, gun readers, scanners, slot readers, and cameras are among the many devices used to read the barcodes.

To help ensure that the data extracted was read correctly, checksums were introduced as a crude form of error detection. More recent barcode formats, such as the Aztec 2D barcode developed by

Andy Longacre in 1995 (US patent number US5591956), but now released to the public domain, use redundancy encoding schemes such as Reed-Solomon. Reed Solomon encoding is adequately discussed in [28], [30] and [34]. The reader is advised to refer to these sources for background information. Very often the degree of redundancy encoding is user selectable.

5   More recently there has also been a move from the simple one dimensional barcodes (line based) to two dimensional barcodes. Instead of storing the information as a series of lines, where the data can be extracted from a single dimension, the information is encoded in two dimensions. Just as with the original barcodes, the 2D barcode contains both information and structural components for better optical recognition. Figure 178 shows an example of a QR Code (Quick

10   Response Code), developed by Denso of Japan (US patent number US5726435). Note the barcode cell is comprised of two areas: a data area (depends on the data being stored in the barcode), and a constant position detection pattern. The constant position detection pattern is used by the reader to help locate the cell itself, then to locate the cell boundaries, to allow the reader to determine the original orientation of the cell (orientation can be determined by the fact

15   that there is no 4th corner pattern).

The number of barcode encoding schemes grows daily. Yet very often the hardware for producing these barcodes is specific to the particular barcode format. As printers become more and more embedded, there is an increasing desire for real-time printing of these barcodes. In particular, Netpage enabled applications require the printing of 2D barcodes (or tags) over the page,

20   preferably in infra-red ink. The tag encoder in SoPEC uses a generic barcode format encoding scheme which is particularly suited to real-time printing. Since the barcode encoding format is generic, the same rendering hardware engine can be used to produce a wide variety of barcode formats.

Unfortunately the term "barcode" is interpreted in different ways by different people. Sometimes it

25   refers only to the data area component, and does not include the constant position detection pattern. In other cases it refers to both data and constant position detection pattern.

We therefore use the term *tag* to refer to the combination of data and any other components (such as position detection pattern, blank space etc. surround) that must be rendered to help hold or locate/read the data. A tag therefore contains the following components:

30   •   data area(s). The data area is the whole reason that the tag exists. The tag data area(s) contains the encoded data (optionally redundancy-encoded, perhaps simply checksummed) where the bits of the data are placed within the data area at locations specified by the tag encoding scheme.

•   constant background patterns, which typically includes a constant position detection

35   pattern. These help the tag reader to locate the tag. They include components that are easy to locate and may contain orientation and perspective information in the case of 2D tags. Constant background patterns may also include such patterns as a blank area surrounding the data area or position detection pattern. These blank patterns can aid in the decoding of the data by ensuring that there is no interference between tags or data areas.

In most tag encoding schemes there is at least some constant background pattern, but it is not necessarily required by all. For example, if the tag data area is enclosed by a physical space and the reading means uses a non-optical location mechanism (e.g. physical alignment of surface to data reader) then a position detection pattern is not required.

5   Different tag encoding schemes have different sized tags, and have different allocation of physical tag area to constant position detection pattern and data area. For example, the QR code has 3 fixed blocks at the edges of the tag for position detection pattern (see Figure 178) and a data area in the remainder. By contrast, the Netpage tag structure (see Figures 179 and 180) contains a circular locator component, an orientation feature, and several data areas. Figure 179(a) shows

10   the Netpage tag constant background pattern in a resolution independent form. Figure 179(b) is the same as Figure 179(a), but with the addition of the data areas to the Netpage tag. Figure 180 is an example of dot placement and rendering to 1600 dpi for a Netpage tag. Note that in Figure 180 a single bit of data is represented by many physical output dots to form a block within the data area.

15   26.2.1   Contents of the data area

The data area contains the data for the tag.

Depending on the tag's encoding format, a single bit of data may be represented by a number of physical printed dots. The exact number of dots will depend on the output resolution and the target reading/scanning resolution. For example, in the QR code (see Figure 178), a single bit is

20   represented by a dark module or a light module, where the exact number of dots in the dark module or light module depends on the rendering resolution and target reading/scanning resolution. For example, a dark module may be represented by a square block of printed dots (all on for binary 1, or all off for binary 0), as shown in Figure 181.

The point to note here is that a single bit of data may be represented in the printed tag by an

25   arbitrary printed shape. The smallest shape is a single printed dot, while the largest shape is theoretically the whole tag itself, for example a giant *macrodot* comprised of many printed dots in both dimensions.

An ideal generic tag definition structure allows the generation of an arbitrary printed shape from each bit of data.

30   26.2.2   What do the bits represent?

Given an original number of bits of data, and the desire to place those bits into a printed tag for subsequent retrieval via a reading/scanning mechanism, the original number of bits can either be placed directly into the tag, or they can be redundancy-encoded in some way. The exact form of redundancy encoding will depend on the tag format.

35   The placement of data bits within the data area of the tag is directly related to the redundancy mechanism employed in the encoding scheme. The idea is generally to place data bits together in 2D so that burst errors are averaged out over the tag data, thus typically being correctable. For example, all the bits of Reed-Solomon codeword would be spread out over the entire tag data area so to minimize being affected by a burst error.

Since the data encoding scheme and shape and size of the tag data area are closely linked, it is desirable to have a generic tag format structure. This allows the same data structure and rendering embodiment to be used to render a variety of tag formats.

26.2.2.1 Fixed and variable data components

In many cases, the tag data can be reasonably divided into fixed and variable components. For example, if a tag holds N bits of data, some of these bits may be fixed for all tags while some may vary from tag to tag.

For example, the Universal product code allows a country code and a company code. Since these bits don't change from tag to tag, these bits can be defined as fixed, and don't need to be provided to the tag encoder each time, thereby reducing the bandwidth when producing many tags.

Another example is Netpage tags. A single printed page contains a number of Netpage tags. The page-id will be constant across all the tags, even though the remainder of the data within each tag may be different for each tag. By reducing the amount of variable data being passed to SoPEC's tag encoder for each tag, the overall bandwidth can be reduced.

Depending on the embodiment of the tag encoder, these parameters will be either implicit or explicit, and may limit the size of tags renderable by the system. For example, a software tag encoder may be completely variable, while a hardware tag encoder such as SoPEC's tag encoder may have a maximum number of tag data bits.

26.2.2.2 Redundancy-encode the tag data within the tag encoder

Instead of accepting the complete number of TagData bits encoded by an external encoder, the tag encoder accepts the basic non-redundancy-encoded data bits and encodes them as required for each tag. This leads to significant savings of bandwidth and on-chip storage.

In SoPEC's case for Netpage tags, only 120 bits of original data are provided per tag, and the tag encoder encodes these 120 bits into 360 bits. By having the redundancy encoder on board the tag encoder the effective bandwidth and internal storage required is reduced to only 33% of what would be required if the encoded data was read directly.

26.3 PLACEMENT OF TAGS ON A PAGE

The TE places tags on the page in a triangular grid arrangement as shown in Figure 182.

The triangular mesh of tags combined with the restriction of no overlap of columns or rows of tags means that the process of tag placement is greatly simplified. For a given line of dots, all the tags on that line correspond to the same part of the general tag structure. The triangular placement can be considered as alternative lines of tags, where one line of tags is inset by one amount in the dot dimension, and the other line of dots is inset by a different amount. The dot inter-tag gap is the same in both lines of tag, and is different from the line inter-tag gap.

Note also that as long as the tags themselves can be rotated, portrait and landscape printing are essentially the same - the placement parameters of line and dot are swapped, but the placement mechanism is the same.

The general case for placement of tags therefore relies on a number of parameters, as shown in Figure 183.

The parameters are more formally described in Table 169. Note that these are placement parameters and not registers.

Table 169. Tag placement parameters

parameter	description	restrictions
Tag height	The number of dot lines in a tag's bounding box	minimum 1
Tag width	The number of dots in a single line of the tag's bounding box. The number of dots in the tag itself may vary depending on the shape of the tag, but the number of dots in the bounding box will be constant (by definition).	minimum 1
Dot inter-tag gap	The number of dots from the edge of one tag's bounding box to the start of the next tag's bounding box, in the dot direction.	minimum = 0
Line inter-tag gap	The number of dot lines from the edge of one tag's bounding box to the start of the next tag's bounding box, in the line direction.	minimum = 0
Start Position	Defines the status of the top left dot on the page - is an offset in dot & row within the tag or the inter-tag gap.	-
AltTagLinePositi on	Defines the status for the start of the alternate row of tags. Is an offset in dot within the tag or within the dot inter-tag gap (the row position is always 0).	-

5   26.4   BASIC TAG ENCODING PARAMETERS

SoPEC's tag encoder imposes range restrictions on tag encoding parameters as a direct result of on-chip buffer sizes. Table 170 lists the basic encoding parameters as well as range restrictions where appropriate. Although the restrictions were chosen to take the most likely encoding scenarios into account, it is a simple matter to adjust the buffer sizes and corresponding

10   addressing to allow arbitrary encoding parameters in future implementations.

Table 170. Encoding parameters

name	definition	maximum value imposed by TE
W	page width	2^{14} dotpairs or 20.48 inches at 1600 dpi
S	tag size	typical tag size is 2mm × 2mm maximum tag size is 384 dots x 384 dots before scaling i.e. 6 mm x 6 mm at 1600 dpi
N	number of dots in each dimension of the tag	384 dots before scaling
E	redundancy encoding for tag data	Reed-Solomon GF(2^4) at 5:10 or 7:8
D_F	size of fixed data (unencoded)	40 or 56 bits
R_F	size of redundancy-encoded fixed	120 bits

	data	
D_V	size of variable data (unencoded)	120 or 112 bits
R_V	size of redundancy-encoded variable data	360 or 240 bits
T	tags per page width	256

The fixed data for the tags on a page need only be supplied to the TE once. It can be supplied as 40 or 56 bits of unencoded data and encoded within the TE as described in Section 26.4.1. Alternatively it can be supplied as 120 bits of pre-encoded data (encoded arbitrarily).

The variable data for the tags on a page are those 112 or 120 data bits that are variable for each tag. Variable tag data is supplied as part of the band data, and is always encoded by the TE as described in Section 26.4.1, but may itself be arbitrarily pre-encoded.

26.4.1   Redundancy encoding

The mapping of data bits (both fixed and variable) to redundancy encoded bits relies heavily on the method of redundancy encoding employed. Reed-Solomon encoding was chosen for its ability to deal with burst errors and effectively detect and correct errors using a minimum of redundancy. Reed Solomon encoding is adequately discussed in [28], [30] and [34]. The reader is advised to refer to these sources for background information.

In this implementation of the TE we use Reed-Solomon encoding over the Galois Field $GF(2^4)$. Symbol size is 4 bits. Each codeword contains 15 4-bit symbols for a codeword length of 60 bits. The primitive polynomial is $p(x) = x^4 + x + 1$, and the generator polynomial is $g(x) = (x+\alpha)(x+\alpha^2)...(x+\alpha^{2t})$, where t = the number of symbols that can be corrected.

Of the 15 symbols, there are two possibilities for encoding:

- RS(15, 5): 5 symbols original data (20 bits), and 10 redundancy symbols (40 bits). The 10 redundancy symbols mean that we can correct up to 5 symbols in error. The generator polynomial is therefore $g(x) = (x+\alpha)(x+\alpha^2)...(x+\alpha^{10})$.

- RS(15, 7): 7 symbols original data (28 bits), and 8 redundancy symbols (32 bits). The 8 redundancy symbols mean that we can correct up to 4 symbols in error. The generator polynomial is $g(x) = (x+\alpha)(x+\alpha^2)...(x+\alpha^8)$.

In the first case, with 5 symbols of original data, the total amount of original data per tag is 160 bits (40 fixed, 120 variable). This is redundancy encoded to give a total amount of 480 bits (120 fixed, 360 variable) as follows:

- Each tag contains up to 40 bits of fixed original data. Therefore 2 codewords are required for the fixed data, giving a total encoded data size of 120 bits. Note that this fixed data only needs to be encoded once per page.

- Each tag contains up to 120 bits of variable original data. Therefore 6 codewords are required for the variable data, giving a total encoded data size of 360 bits.

In the second case, with 7 symbols of original data, the total amount of original data per tag is 168 bits (56 fixed, 112 variable). This is redundancy encoded to give a total amount of 360 bits (120 fixed, 240 variable) as follows:

- Each tag contains up to 56 bits of fixed original data. Therefore 2 codewords are required for the fixed data, giving a total encoded data size of 120 bits. Note that this fixed data only needs to be encoded once per page.
- Each tag contains up to 112 bits of variable original data. Therefore 4 codewords are required for the variable data, giving a total encoded data size of 240 bits.

The choice of data to redundancy ratio depends on the application.

26.5 DATA STRUCTURES USED BY TAG ENCODER

26.5.1 Tag Format Structure

The Tag Format Structure (TFS) is the template used to render tags, optimized so that the tag can be rendered in real time. The TFS contains an entry for each dot position within the tag's bounding box. Each entry specifies whether the dot is part of the constant background pattern or part of the tag's data component (both fixed and variable).

The TFS is very similar to a bitmap in that it contains one entry for each dot position of the tag's bounding box. The TFS therefore has *TagHeight x TagWidth* entries, where *TagHeight* matches the height of the bounding box for the tag in the line dimension, and *TagWidth* matches the width of the bounding box for the tag in the dot dimension. A single line of TFS entries for a tag is known as a *tag line structure*.

The TFS consists of *TagHeight* number of *tag line structures*, one for each 1600 dpi line in the tag's bounding box. Each tag line structure contains three contiguous tables, known as tables A, B, and C. Table A contains 384 2-bit entries, one entry for each of the maximum number of dots in a single line of a tag (see Table). The actual number of entries used should match the size of the bounding box for the tag in the dot dimension, but all 384 entries must be present. Table B contains 32 9-bit data addresses that refer to (in order of appearance) the data dots present in the particular line. All 32 entries must be present, even if fewer are used. Table C contains two 5-bit pointers into table B, and therefore comprises 10 bits. Padding of 214 bits is added. The total length of each tag line structure is therefore 5 × 256-bit DRAM words. Thus a TFS containing *TagHeight* tag line structures requires a *TagHeight* * 160 bytes. The structure of a TFS is shown in Figure 184.

A full description of the interpretation and usage of Tables A, B and C is given in section 26.8.3 on page 444.

26.5.1.1 Scaling a tag

If the size of the printed dots is too small, then the tag can be scaled in one of several ways. Either the tag itself can be scaled by N dots in each dimension, which increases the number of entries in the TFS. As an alternative, the output from the TE can be scaled up by pixel replication via a scale factor greater than 1 in the both the TE and TFU.

For example, if the original TFS was 21 × 21 entries, and the scaling were a simple 2 × 2 dots for each of the original dots, we could increase the TFS to be 42 × 42. To generate the new TFS from the old, we would repeat each entry across each line of the TFS, and then we would repeat each line of the TFS. The net number of entries in the TFS would be increased fourfold (2 × 2).

The TFS allows the creation of *macrodots* instead of simple scaling. Looking at Figure 185 for a simple example of a 3 × 3 dot tag, we may want to produce a physically large printed form of the tag, where each of the original dots was represented by 7 × 7 printed dots. If we simply performed replication by 7 in each dimension of the original TFS, either by increasing the size of the TFS by 7 in each dimension or putting a scale-up on the output of the tag generator output, then we would have 9 sets of 7 × 7 square blocks. Instead, we can replace each of the original dots in the TFS by a 7 × 7 dot definition of a rounded dot. Figure 186 shows the results.

Consequently, the higher the resolution of the TFS the more printed dots can be printed for each *macrodot*, where a macrodot represents a single data bit of the tag. The more dots that are available to produce a macrodot, the more complex the pattern of the macrodot can be. As an example, Figure n page461 on page **Error! Bookmark not defined.** shows the Netpage tag structure rendered such that the data bits are represented by an average of 8 dots × 8 dots (at 1600 dpi), but the actual shape structure of a dot is not square. This allows the printed Netpage tag to be subsequently read at any orientation.

26.5.2   Raw tag data

The TE requires a band of unencoded variable tag data if variable data is to be included in the tag bit-plane. A band of unencoded variable tag data is a set of contiguous unencoded tag data records, in order of encounter top left of printed band from top left to lower right.

An unencoded tag data record is 128 bits arranged as follows: bits 0-111 or 0-119 are the bits of raw tag data, bit 120 is a flag used by the TE (*TagIsPrinted*), and the remaining 7 bits are reserved (and should be 0). Having a record size of 128 bits simplifies the tag data access since the data of two tags fits into a 256-bit DRAM word. It also means that the flags can be stored apart from the tag data, thus keeping the raw tag data completely unrestricted. If there is an odd number of tags in line then the last DRAM read will contain a tag in the first 128 bits and padding in the final 128 bits.

The *TagIsPrinted* flag allows the effective specification of a tag resolution mask over the page. For each tag position the *TagIsPrinted* flag determines whether any of the tag is printed or not. This allows arbitrary placement of tags on the page. For example, tags may only be printed over particular active areas of a page. The *TagIsPrinted* flag allows only those tags to be printed.

TagIsPrinted is a 1 bit flag with values as shown in Table 171.

Table 171. TagIsPrinted values

Value	description
0	Don't print the tag in this tag position. Output 0 for each dot within the tag bounding box.
1	Print the tag as specified by the various tag structures.

26.5.3   DRAM storage requirements

The total DRAM storage required by a single band of raw tag data depends on the number of tags present in that band. Each tag requires 128 bits. Consequently if there are *N* tags in the band, the size in DRAM is *16N* bytes.

The maximum size of a line of tags is 163 × 128 bits. When maximally packed, a row of tags contains 163 tags (see Table) and extends over a minimum of 126 print lines. This equates to 282 KBytes over a Letter page.

The total DRAM storage required by a single TFS is *TagHeight*/7 KBytes (including padding).

5   Since the likely maximum value for *TagHeight* is 384 (given that SoPEC restricts *TagWidth* to 384), the maximum size in DRAM for a TFS is 55 KBytes.

26.5.4   DRAM access requirements

The TE has two separate read interfaces to DRAM for raw tag data, TD, and tag format structure, TFS.

10   The memory usage requirements are shown in Table 172. Raw tag data is stored in the compressed page store

Table 172. Memory usage requirements

Block	Size	Description
Compressed page store	2048 Kbytes	Compressed data page store for Bi-level, contone and raw tag data.
Tag Format Structure	55 Kbyte (384 dot line tags @ 1600 dpi)	55 kB in PEC1 for 384 dot line tags (the benchmark) at 1600 dpi 2.5 mm tags (1/10th inch) @ 1600 dpi require 160 dot lines = 160/384 x55 or 23 kB 2.5 mm tags @ 800 dpi require 80/384 x55 = 12 kB

15   The TD interface will read 256-bits from DRAM at a time. Each 256-bit read returns 2 times 128-bit tags. The TD interface to the DIU will be a 256-bit double buffer. If there is an odd number of tags in line then the last DRAM read will contain a tag in the first 128 bits and padding in the final 128 bits.

The TFS interface will also read 256-bits from DRAM at a time. The TFS required for a line is 136
20   bytes. A total of 5 times 256-bit DRAM reads is required to read the TFS for a line with 192 unused bits in the fifth 256-bit word. A 136-byte double-line buffer will be implemented to store the TFS data.

The TE's DIU bandwidth requirements are summarized in Table 173.

Table 173. DRAM bandwidth requirements

25

Block Name	Direction	Maximum number of cycles between each 256-bit DRAM access	Peak Bandwidth (bits/cycle)	Average Bandwidth (bits/cycle)

TD	Read	Single 256 bit reads1.	1.02	1.02
TFS	Read	Single 256 bit reads2. TFS is 136 bytes. This means there is unused data in the fifth 256 bit read. A total of 5 reads is required.	0.093	0.093

1: Each 2mm tag lasts 126 dot cycles and requires 128 bits. This is a rate of 256 bits every 252 cycles.

2: 17 x 64 bit reads per line in PEC1 is 5 x 256 bit reads per line in SoPEC with unused bits in the last 256-bit read.

26.5.5   TD and TFS Bandstore wrapping

Table 174. Bandstore Inputs from CDU

Port Name	Pins	I/O	Description
cdu_endofbandstore[21: 5]	17	In	Address of the end of the current band of data. 256-bit word aligned DRAM address.
cdu_startofbandstore[21: 5]	17	In	Address of the start of the current band of data. 256-bit word aligned DRAM address.

Both TD and TFS storage in DRAM can wrap around the bandstore area. The bounds of the band store are described by inputs from the CDU shown in Table 174. The TD and TFS DRAM interfaces therefore support bandstore wrapping. If the TD or TFS DRAM interface increments an address it is checked to see if it matches the end of bandstore address. If so, then the address is mapped to the start of the bandstore.

26.5.6   Tag sizes

SoPEC allows for tags to be between 0 to 384 dots. A typical 2 mm tag requires 126 dots. Short tags do not change the internal bandwidth or throughput behaviours at all. Tag height is specified so as to allow the DRAM storage for raw tag data to be specified. Minimum tag width is a condition imposed by throughput limitations, so if the width is too small TE cannot consistently produce 2 dots per cycle across several tags (also there are raw tag data bandwidth implications). Thinner tags still work, they just take longer and/or need scaling.

26.6   IMPLEMENTATION

26.6.1   Tag Encoder Architecture

A block diagram of the TE can be seen below.

The TE writes lines of bi-level tag plane data to the TFU for later reading by the HCU. The TE is responsible for merging the encoded tag data with the tag structure (interpreted from the TFS). Y-integer scaling of tags is performed in the TE with X-integer scaling of the tags performed in the TFU. The encoded tag layer is generated 2 bits at a time and output to the TFU at this rate. The HCU however only consumes 1 bit per cycle from the TFU. The TE must provide support for 126dot Tags (2mm densely packed) with 108 Tags per line with 128bits per tag.

413

The tag encoder consists of a TFS interface that loads and decodes TFS entries, a tag data interface that loads tag raw data, encodes it, and provides bit values on request, and a state machine to generate appropriate addressing and control signals. The TE has two separate read interfaces to DRAM for raw tag data, TD, and tag format structure, TFS.

5   It is possible that the raw tag data interface, the TD, to the DIU could be replaced by a hardware state machine at a later stage. This would allow flexibility in the generation of tags. Support for Y scaling needs to be added to the PEC1 TE. The PEC1 TE already allows stalling at its output during a line when *tfu_te_oktowrite* is deasserted.

26.6.2   Y-Scaling output lines

10   In order to support scaling in the Y direction the following modifications to the PEC1 TE are suggested to the Tag Data Interface, Tag Format Structure Interface and TE Top Level:

- for Tag Data Interface: program the configuration registers of *Table* , *firstTagLineHeight* and *tagMaxLine* with true value i.e. not multiplied up by the scale factor *YScale*. Within the
15   Tag Data interface there are two counters, *countx* and *county* that have a direct bearing on the *rawTagDataAddr* generation. *countx* decrements as tags are read from DRAM. It is reset to *NumTags[RtdTagSense]* at start of each line of tags. *county* is decremented as each line of tags is completely read from DRAM i.e. *countx* = 0. Scaling may be performed by counting the number of times *countx* reaches zero and only decrementing *county* when
20   this number reaches *YScale*. This will cause the TagData Interface to read each line of tag data *NumTags[RtdTagSense]* * *YScale* times.
- for Tag Format Structure Interface: The implication of Y-scaling for the TFS is that each Tag Line Structure is used *YScale* times. This may be accomplished in either of two ways:
- For each Tag Line Structure read it once from DRAM and reuse *YScale* times. This
25   involves gating the control of TFS buffer flipping with *YScale*. Because of the way in which this *advTfsLine* and *advTagLine* related functionality is coded in the PEC1 TFS this solution is judged to be error-prone.
- Fetch each TagLineStructure *YScale* times. This solution involves controlling the activity of *currTfsAddr* with *YScale*.
30   In SoPEC the TFS must supply five addresses to the DIU to read each individual Tag Line Structure. The DIU returns 4*64-bit words for each of the 5 accesses. This is different from the behaviour in PEC1, where one address is given and 17 data-words were returned by the DIU.

Since the behaviour of the *currTfsAddr* must be changed to meet the requirements of the
35   SoPEC DIU it makes sense to include the Y-Scaling into this change i.e. a count of the number of completed sets of 5 accesses to the DIU is compared to YScale. Only when this count equals YScale can *currTfsAddr* be loaded with the base address of the next lines Tag Line Structure in DRAM, otherwise it is re-loaded with the base address of the current lines Tag Line Structure in DRAM.

414

• For Top Level: The Top Level of the TE has a counter, *LinePos*, which is used to count the number of completed output lines when in a tag gap or in a line of tags. At the start (i.e. top-left hand dot-pair) of a gap or tag *LinePos* is loaded with either *TagGapLine* or *TagMaxLine*. The value of *LinePos* is decremented at last dot-pair in line. Y-Scaling may be accomplished by gating the decrement of *LinePos* based on *YScale* value

26.6.3   TE Physical Hierarchy

Figure 188 above illustrates the structural hierarchy of the TE. The top level contains the Tag Data Interface (TDI), Tag Format Structure (TFS), and an FSM to control the generation of dot pairs along with a clocked process to carry out the PCU read/write decoding. There is also some additional logic for muxing the output data and generating other control signals.

At the highest level, the TE state machine processes the output lines of a page one line at a time, with the starting position either in an inter-tag gap or in a tag (a SoPEC may be only printing part of a tag due to multiple SoPECs printing a single line).

If the current position is within an inter-tag gap, an output of 0 is generated. If the current position is within a tag, the tag format structure is used to determine the value of the output dot, using the appropriate encoded data bit from the fixed or variable data buffers as necessary. The TE then advances along the line of dots, moving through tags and inter-tag gaps according to the tag placement parameters.

26.6.4   IO Definitions

Table 175. TE Port List

Port Name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	SoPEC Functional clock.
prst_n	1	In	Global reset signal.
Bandstore Signals			
cdu_endofbandstore[21:5]	17	In	Address of the end of the current band of data. 256-bit word aligned DRAM address.
cdu_startofbandstore[21:5]	17	In	Address of the start of the current band of data. 256-bit word aligned DRAM address.
te_finishedband	1	Out	TE finished band signal to PCU and ICU.
PCU Interface data and control signals			
pcu_addr[8:2]	7	In	PCU address bus. 7 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
te_pcu_datain[31:0]	32	Out	Read data bus from the TE to the PCU.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_te_sel	1	In	Block select from the PCU. When *pcu_te_sel* is high both *pcu_addr* and *pcu_dataout* are valid.

te_pcu_rdy	1	Out	Ready signal to the PCU. When *te_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *te_pcu_datain* is valid.
TD (raw Tag Data) DIU Read Interface signals			
td_diu_rreq	1	Out	TD requests DRAM read. A read request must be accompanied by a valid read address.
td_diu_radr[21:5]	17	Out	TD read address to DIU. 17 bits wide (256-bit aligned word).
diu_td_rack	1	In	Acknowledge from DIU that TD read request has been accepted and new read address can be placed on *te_diu_radr*.
diu_data[63:0]	64	In	Data from DIU to TE. First 64-bits are bits 63:0 of 256 bit word; Second 64-bits are bits 127:64 of 256 bit word; Third 64-bits are bits 191:128 of 256 bit word; Fourth 64-bits are bits 255:192 of 256 bit word.
diu_td_rvalid	1	In	Signal from DIU telling TD that valid read data is on the *diu_data* bus.
TFS (Tag Format Structure) DIU Read Interface signals			
tfs_diu_rreq	1	Out	TFS requests DRAM read. A read request must be accompanied by a valid read address.
tfs_diu_radr[21:5]	17	Out	TFS Read address to DIU 17 bits wide (256-bit aligned word).
diu_tfs_rack	1	In	Acknowledge from DIU that TFS read request has been accepted and new read address can be placed on *tfs_diu_radr*.
diu_data[63:0]	64	In	Data from DIU to TE. First 64-bits are bits 63:0 of 256 bit word; Second 64-bits are bits 127:64 of 256 bit word; Third 64-bits are bits 191:128 of 256 bit word; Fourth 64-bits are bits 255:192 of 256 bit word.
diu_tfs_rvalid	1	In	Signal from DIU telling TFS that valid read data is on the *diu_data* bus.
TFU Interface data and control signals			
tfu_te_oktowrite	1	In	Ready signal indicating TFU has space available and is ready to be written to. Also asserted from the point that the TFU has recieved its expected

			number of bytes for a line until the next *te_tfu_wradvline*
te_tfu_wdata[7:0]	8	Out	Write data for TFU.
te_tfu_wdatavalid	1	Out	Write data valid signal. This signal remains high whenever there is valid output data on *te_tfu_wdata*
te_tfu_wradvline	1	Out	Advance line signal strobed when the last byte in a line is placed on *te_tfu_wdata*

26.6.5 Configuration Registers

The configuration registers in the TE are programmed via the PCU interface.Refer to section 21.8.2 on page 321 for the description of the protocol and timing diagrams for reading and writing registers in the TE.Note that since addresses in SoPEC are byte aligned and the PCU only supports 32-bit register reads and writes the lower 2 bits of the PCU address bus are not required to decode the address space for the TE.Table 176 lists the configuration registers in the TE. Registers which address DRAM are 64-bit DRAM word aligned as this is the case for the PEC1 TE. SoPEC assumes a 256-bit DRAM word size. If the TE can be easily modified then the DRAM word addressing should be modified to 256-bit word aligned addressing. Otherwise, software should program these the 64-bit word aligned addresses on a 256-bit DRAM word boundary..

Table 176. TE Configuration Registers

Address TE_base+	register name	#bits	value on reset	description
Control registers				
0x00	Reset	1	1	A write to this register causes a reset of the TE. This register can be read to indicate the reset state: 0 - reset in progress 1 - reset not in progress
0x04	Go	1	0	Writing 1 to this register starts the TE. Writing 0 to this register halts the TE. When *Go* is deasserted the state-machines go to their idle states but all counters and configuration registers keep their values. When *Go* is asserted all counters are reset, but con-

				figuration registers keep their values (i.e. they don't get reset). *NextBandEnable* is cleared when *Go* is asserted. The TFU must be started before the TE is started. This register can be read to determine if the TE is running (1 = running, 0 = stopped).
Setup registers (constant for processing of a page)				
0x40	TfsStartAdr (64-bit aligned DRAM address - should start at a 256-bit aligned loca- tion)	19	0	Points to the first word of the first TFS line in DRAM.
0x44	TfsEndAdr (64-bit aligned DRAM address - should start at a 256-bit aligned loca- tion)	19	0	Points to the first word of the last TFS line in DRAM.
0x48	TfsFirstLineA dr (64-bit aligned DRAM address)	19	0	Points to the first word of the first TFS line to be encountered on the page. If the start of the page is in an inter-tag gap, then this value will be the same as *TFSStartAdr* since the first tag line reached will be the top line of a tag.

0x4C	DataRedun	1	0	Defines the data to redundancy ratio for the Reed Solomon encoder. Symbol size is always 4 bits, Code-word size is always 15 symbols (60 bits). 0 - 5 data symbols (20 bits), 10 redundancy symbols (40 bits) 1 -7 data symbols (28 bits), 8 redundancy symbols (32 bits)
0x50	Decode2DEn	1	0	Determines whether or not the data bits are to be 2D decoded rather than redundancy encoded (each 2 bits of the data bits becomes 4 output data bits). 0 = redundancy encode data 1 = decode each 2 bits of data into 4 bits
0x54	VariableData Present	1	0	Defines whether or not there is variable data in the tags. If there is none, no attempt is made to read tag data, and tag encoding should only reference fixed tag data.
0x58	EncodeFixed	1	0	Determines whether or not the lower 40 (or 56) bits of fixed data should be encoded into 120 bits or simply used as is.
0x5C	TagMaxDotpairs	8	0	The width of a tag in dot-pairs, minus 1. Minimum 0, Maximum=191.
0x60	TagMaxLine	9	0	The number of lines in a tag, minus 1. Minimum 0, Maximum = 383.
0x64	TagGapDot	14	0	The number of dot pairs between tags in the dot

				dimension minus 1. Only valid if *TagGapPresent*[bit 0] = 1.
0x68	TagGapLine	14	0	Defines the number of dotlines between tags in the line dimension minus 1. Only valid if *TagGapPresent*[bit1] = 1.
0x6C	DotPairsPerLine	14	0	Number of output dot pairs to generate per tag line.
0x70	DotStartTagSense	2	0	Determines for the first/even (bit 0) and second/odd (bit 1) rows of tags whether or not the first dot position of the line is in a tag. 1 = in a tag, 0 = in an inter-tag gap.
0x74	TagGapPresent	2	0	Bit 0 is 1 if there is an inter- tag gap in the dot dimension, and 0 if tags are tightly packed. Bit 1 is 1 if there is an inter- tag gap in the line dimension, and 0 if tags are tightly packed.
0x78	YScale	8	1	Tag scale factor in Y direction. Output lines to the TFU will be generated YScale times.
0x80 to 0x84	DotStartPos	2x14	0	Determines for the first/even (0) and second/odd (1) rows of tags the number of dotpairs remaining minus 1, in either the tag or inter-tag gap at the start of the line.
0x88 to 0x8C	NumTags	2x8	0	Determines for the first/even and second/odd rows of tags how many tags are present in a line (equals number of tags

				minus 1).
Setup band related registers				
0xC0	NextBandStartTagDataAdr (64-bit aligned DRAM address - should start at a 256-bit aligned loca-tion)			Holds the value of StartTagDataAdr for the next band. This value is copied to StartTagDataAdr when DoneBand is 1 and NextBandEnable is 1, or when *Go* transitions from 0 to 1.
0xC4	NextBandEndOfTagData (64-bit aligned DRAM address)			Holds the value of EndOfTagData for the next band. This value is copied to EndOfTagData when DoneBand is 1 and NextBandEnable is 1, or when *Go* transitions from 0 to 1.
0xC8	NextBandFirstTagLine-Height	9	0	Holds the value of FirstTagLineHeight for the next band. This value is copied to FirstTagLineHeight when DoneBand gets is 1 and NextBandEnable is 1, or when *Go* transitions from 0 to 1.
0xCC	NextBandEnable			When NextBandEnable is 1 and DoneBand is 1, then when te_finishedband is set at the end of a band: -NextBandStartTagDataAdr is copied to StartTagDataAdr -NextBandEndOfTagData is copied to EndOfTagData -NextBandFirstTagLineHeight is copied to FirstTa-

				gLineHeight
				-DoneBand is cleared
				-NextBandEnable is cleared.
				NextBandEnable is cleared when *Go* is asserted.
Read-only band related registers				
0xD0	DoneBand	1	0	Specifies whether the tag data interface has finished loading all the tag data for the band. It is cleared to 0 when *Go* transitions from 0 to 1. When the tag data interface has finished loading all the tag data for the band, the *te_finishedband* signal is given out and the *DoneBand* flag is set. If *NextBandEnable* is 1 at this time then *startTagDataAdr*, *endOfTagData* and *firstTaglineHeight* are updated with the values for the next band and *DoneBand* is cleared. Processing of the next band starts immediately. If *NextBandEnable* is 0 then the remainder of the TE will continue to run,, while the read control unit waits for *NextBandEnable* to be set before it restarts. Read only.
0xD4	StartTagData Adr (64-bit aligned DRAM address - should start at	19	0	The start address of the current row of raw tag data. This is initially points to the first word of the band's tag data, which should be aligned to a 128-bit boundary (i.e. the lower bit of this address

	a 256-bit aligned loca- tion)			should be 0). Read only.
0xD8	EndOfTagDat a (64-bit aligned DRAM address)	19	0	Points to the address of the final tag for the band. When all the tag data up to and including address *endOfTagData* has been read in, the *te_finishedband* signal is given and the doneBand flag is set. Read only.
0xDC	FirstTagLineH eight	9	0	The number of lines minus 1 in the first tag encountered in this band. This will be equal to *TagMaxLine* if the band starts at a tag boundary. Read only.
Work registers (set before starting the TE and must not be touched between bands)				
0x100	LineInTag	1	0	Determines whether or not the first line of the page is in a line of tags or in an inter-tag gap. 1 - in a tag, 0 - in an inter-tag gap.
0x104	LinePos	14	0	The number of lines remaining minus 1, in either the tag or the inter-tag gap in at the start of the page.
0x110 to 0x11C	TagData	4x32	0	This 128 bit register must be set up initially with the fixed data record for the page. This is either the lower 40 (or 56) bits (and the *encodeFixed* register should be set), or the lower 120 bits (and

				encodedFixed should be clear). The tagData[0] register contains the lower 32 bits and the tagData[3] register contains the upper 32 bits. This register is used throughout the tag encoding process to hold the next tag's variable data.
Work registers (set internally) Read-only from the point of view of PCU register access				
0x140	DotPos	14	0	Defines the number of dotpairs remaining in either the tag or inter-tag gap. Does not need to be setup.
0x144	CurrTagPlaneAdr	14	0	The dot-pair number being generated.
0x148	DotsInTag	1	0	Determines whether the current dot pair is in a tag or not 1 - in a tag, 0 - in an inter-tag gap.
0x14C	TagAltSense	1	0	Determines whether the production of output dots is for the first (and subsequent even) or second (and subsequent odd) row of tags.
0x154	CurrTFSAdr (64-bit aligned DRAM address)	19	0	Points to the start next line of the TFS to be read in.
0x158	ReadsRemaining	4	0	Number of reads remaining in the current burst from the raw tag data interface

0x15C	CountX	8	0	The number of tags remaining to be read (minus 1) by the raw tag data interface for the current line.
0x160	CountY	9	0	The number of times (minus 1) the tag data for the current line of tags needs to be read in by the raw tag data interface.
0x164	RtdTagSense	1	0	Determines whether the raw tag data interface is currently reading even rows of tags (=0) or odd rows of tags (=1) *with respect to the start of the page*. Note that this can be different from tagAltSense since the raw tag data interface is reading ahead of the production of dots.
0x168	RawTagData Adr (64-bit aligned DRAM address)	19	0	The current read address within the unencoded raw tag data.

The PCU accessible registers are divided amongst the TE top level and the TE sub-blocks. This is achieved by including write decoders in the sub-blocks as well as the top level, see Figure 189. In order to perform reads the sub-block registers are fed to the top level where the read decode is carried out on all the PCU accessible TE registers.

26.6.5.1 Starting the TE and restarting the TE between bands

The TE must be started after the TFU.

For the first band of data, users set up *NextBandStartTagDataAdr*, *NextBandEndTagData* and *NextBandFirstTagLineHeight* as well as other TE configuration registers. Users then set the TE's *Go* bit to start processing of the band. When the tag data for the band has finished being decoded, the *te_finishedband* interrupt will be sent to the PCU and ICU indicating that the memory associated with the first band is now free. Processing can now start on the next band of tag data.

425

In order to process the next band *NextBandStartTagDataAdr*, *NextBandEndTagData* and *NextBandFirstTagLineHeight* need to be updated before writing a 1 to *NextBandEnable*. There are 4 mechanisms for restarting the TE between bands:

a. *te_finishedband* causes an interrupt to the CPU. The TE will have set its *DoneBand* bit. The CPU reprograms the *NextBandStartTagDataAdr*, *NextBandEndTagData* and *NextBandFirstTagLineHeight* registers, and sets *NextBandEnable* to restart the TE.

b. The CPU programs the TE's *NextBandStartTagDataAdr*, *NextBandEndTagData* and *NextBandFirstTagLineHeight* registers and sets the *NextBandEnable* flag before the end of the current band. At the end of the current band the TE sets *DoneBand*. As *NextBandEnable* is already 1, the TE starts processing the next band immediately.

c. The PCU is programmed so that *te_finishedband* triggers the PCU to execute commands from DRAM to reprogram the *NextBandStartTagDataAdr*, *NextBandEndTagData* and *NextBandFirstTagLineHeight* registers and set the *NextBandEnable* bit to start the TE processing the next band. The advantage of this scheme is that the CPU could process band headers in advance and store the band commands in DRAM ready for execution.

d. This is a combination of *b* and *c* above. The PCU (rather than the CPU in *b*) programs the TE's *NextBandStartTagDataAdr, NextBandEndTagData* and *NextBandFirstTagLineHeight* registers and sets the *NextBandEnable* bit before the end of the current band. At the end of the current band the TE sets *DoneBand* and pulses *te_finishedband*. As *NextBandEnable* is already 1, the TE starts processing the next band immediately. Simultaneously, *te_finishedband* triggers the PCU to fetch commands from DRAM. The TE will have restarted by the time the PCU has fetched commands from DRAM. The PCU commands program the TE next band shadow registers and sets the *NextBandEnable* bit.

After the first tag on the page, all bands have their first tag start at the top i.e. *NextBandFirstTagLineHeight* = *TagMaxLine*. Therefore the same value of *NextBandFirstTagLineHeight* will normally be used for all bands. Certainly, *NextBandFirstTagLineHeight* should not need to change after the second time it is programmed.

26.6.6   TE Top Level FSM

The following diagram illustrates the states in the FSM.

At the highest level, the TE state machine steps through the output lines of a page one line at a time, with the starting position either in an inter-tag gap (signal *dotsintag* = 0) or in a tag (signals *tfsvalid* and *tdvalid* and *lineintag* = 1) (a SoPEC may be only printing part of a tag due to multiple SoPECs printing a single line).

If the current position is within an inter-tag gap, an output of 0 is generated. If the current position is within a tag, the tag format structure is used to determine the value of the output dot, using the appropriate encoded data bit from the fixed or variable data buffers as necessary. The TE then advances along the line of dots, moving through tags and inter-tag gaps according to the tag placement parameters.

Table 177 highlights the signals used within the FSM.

Table 177. Signals used within TE top level FSM

Signal Name	Function
pclk	Sync clock used to register all data within the FSM
prst_n, te_reset	Reset signals
advtagline	1 cycles pulse indicating to TDI and TFS sub-blocks to move onto the next line of Tag data
currdotlineadr[13:0]	Address counter starting 2 pclk ahead of currtagplaneadr to generate the correct dotpair for the current line
dotpos	Counter to identify how many dotpairs wide the tag/gap is
dotsintag	Signal identifying whether the dotpair are in a tag(1)/gap(0)
lineintag_temp	Identical to lineintag but generated 1 pclk earlier
linepos_shadow	Shadow register for linepos due to linepos being written to by 2 different processes
talaltsense	Flag which alternates between tag/gap lines
te_state	FSM state variable
teplanebuf	6-bit shift register used to format dotpairs into a byte for the TFU
wradvline	Advance line signal strobed when the last byte in a line is placed on te_tfu_wdata

Due to the 2 *system clock* delay in the TFS (both Table A and Table B outputs are registered) the TE FSM is working 2 *system clock* cycles AHEAD of the logic generating the write data for the

5   TFU. As a result the following control signals had to be single/double registered on the *system clock*.

The *tag_dot_line* state can be broken down into 3 different stages.

Stage1:- The state *tag_dot_line* is entered due to the *go* signal becoming active. This state controls the writing of dotbytes to the TFU. As long as the tag line buffer address is not equal to

10   the *dotpairsperline* register value and *tfu_te_oktowrite* is active, and there is valid TFS and TD available or taggaps, dotpairs are buffered into bytes and written to the TFU. The tag line buffer address is used internally but not supplied to the TFU since the TFU is a FIFO rather than the line store used in PEC1.

While generating the dotline of a tag/gap line (*lineintag* flag = 1) the dot position counter *dotpos* is

15   decremented/reloaded (with tagmaxdotpairs or taggapdot) as the TE moves between tags/gaps. The *dotsintag* flag is toggled between tags/gaps (0 for a gap, 1 for a tag). This pattern continues until the end of a dotline approaches (*currdotlineadr == dotpairsperline*).

2 *system clock* cycles before the end of the dotline the *lineintag* and *tagaltsense* signals must be prepared for the next dotline be it in a tag/gap dotline or a purely gap dotline.

20   *Stage2*:- At this point the end of a dot line is reached so it is time to decrement the *linepos* counter if still in a tag/gap row or reload the *linepos* register, *dotpos* counter and reprogram the *dotsintag* flag if going onto another tag/gap or pure gap row. Any signal with the _temp extension means this register is updated a cycle early in order for the real register to get its correct value while

427

switching between dot lines and tag rows when *dotpos* and *linepos* counters reach zero i.e when *dotpos* = 0 the end of a tag/gap has been reached, when *linepos* = 0 the end of a tag row is reached. This stage uses the signals *lineintag_temp* and *tagaltsense* which were generated one *system clock* cycle earlier in Stage 1.

5 *Stage3:*- This stage implements the writing of dotpairs to the correct part of the 6-bit shift register based on the LSBs of *currtagplaneadr* and also implements the counter for the *currtagplaneadr*. The *currtagplaneadr* is reset on reaching *currtagplaneadr* = (*dotpairsperline* - 1). All the qualifier signals e.g *dotsintag* for this stage are delayed by 2 *system clock* cycles i.e. the *currtagplaneadr* (which is the internal write address not needed by the TFU) cannot be incremented until the

10 dotpairs are available which is always 2 *system clock* cycles later than when *currdotlineadr* is incremented.

The *wradvline* and *advtagline* pulses are generated using the same logic (currently separated in the PEC1 Tag Encoder VHDL for clarity). Both of these pulses used to update further registers hence the reason they do not use the delayed by 2 *system clock* cycle qualifiers.

15 26.6.7   Combinational Logic

The TDI is responsible for providing the information data for a tag while the TFSI is responsible for deciding whether a particular dot on the tag should be printed as background pattern or tag information. Every dot within a tag's boundary is either an information dot or part of the background pattern.

20 The resulting lines of dots are stored in the TFU.

The TFSI reads one Tag Line Structure (TLS) from the DIU for every dot line of tags. Depending on the current printing position within the tag (indicated by the signal *tagdotnum*), the TFS interface outputs dot information for two dots and if necessary the corresponding read addresses for encoded tag data. The read address are supplied to the TDI which outputs the corresponding

25 data values.

These data values (*tdi_etd0* and *tdi_etd1*) are then combined with the dot information (*tfsi_ta_dot0* and *tfsi_ta_dot1*) to produce the dot values that will actually be printed on the page (*dots*), see Figure 192.

The signal *lastdotintag* is generated by checking that the dots are in a tag (*dotsintag* = 1) and that

30 the dotposition counter dotpos is equal to zero. It is also used by the TFS to load the index address register with zeros at the end of a tag as this is always the starting index when going from one tag to the next. *lastdotintag* is gated with *advtagline* in the TFSi (Table C) where *adv_tfs_line* pulse is used to update the Table C address reg for the new tag line - this is because *lastdotintag* occurs a cycle earlier than *adv_tfs_line* which would result in the wrong Table C value for the last

35 dotpair. *lastdotintag* is also used in the TDi FSM (etd_switch state) to pulse the *etd_advtag* signal hence switching buffers in the ETDi for the next tag.

The signal *lastdotintag1* is identical to *lastdotintag* except it is combinatorially generated (1 cycle earlier than *lastdotintag*, except at the end of a *tagline*). *lastdotintag1* signal is only used in the TDi to reset the *tdvalid* signal on the cycle when *dotpos* = 0. Note the UNSIGNED(*currdotlineadr*) =

UNSIGNED(*dotpairsperline*) - 1 not UNSIGNED(*currdotlineadr*) = UNSIGNED(*dotpairsperline*) - 2 as in the *lastdotintag_gen* process as this is an combinatorial process.

The *dotposvalid* signal is created based on being in a tag line (*lineintag1* = 1), dots being in a tag (*dotsintag1* = 1), having a valid tag format structure available (*tfsvalid1* = 1) and having encoded

5   tag data available (*tdvalid1* = 1). Note that each of the qualifier signals are delayed by 1 *pclk* cycle due to the registering of Table A output data into Table C where *dotposvalid* is used. The *dotposvalid* signal is used as an enable to load the Table C address register with the next index into Table B which in turn provides the 2 addresses to make 2 dots available.

The signal *te_tfu_wdatavalid* can only be active if in a taggap or if valid tag data is available

10   (*tdvalid2* and *tfsvalid2*) and the *currtagpplaneadr*(1:0) equal 11 i.e. a byte of data has been generated by combining four dotpairs.

The signal *tagdotnum* tells the TFS how many dotpairs remain in a tag/gap. It is calculated by subtracting the value in the *dotpos* counter from the value programmed in the *tagmaxdotpairs* register.

15   26.7   TAG DATA INTERFACE (TDI)

26.7.1   I/O Specification

Table 178. TDI Port List

signal name	I/O	Description
Clocks and Resets		
pclk	In	SoPEC system clock
prst_n	In	Active-low, synchronous reset in pclk domain.
DIU Read Interface Signals		
diu_data[63:0]	In	Data from DRAM.
td_diu_rreq	Out	Data request to DRAM.
td_diu_radr[21:5]	Out	Read address to DRAM.
diu_td_rack	In	Data acknowledge from DRAM.
diu_td_rvalid	In	Data valid signal from DRAM.
PCU Interface Data, Control Signals and		
pcu_dataout[31:0]	In	PCU writes this data.
pcu_addr[8:2]	In	PCU accesses this address.
pcu_rwn	In	Global read/write-not signal from PCU.
pcu_te_sel	In	PCU selects TE for r/w access.
pcu_te_reset	In	PCU reset.
td_te_doneband td_te_dataredun td_te_decode2den td_te_variabledatapresent td_te_encodefixed	Out	PCU readable registers.

429

td_te_numtags0		
td_te_numtags1		
td_te_starttagdataadr		
td_te_rawtagdataadr		
td_te_endoftagdata		
td_te_firsttaglineheight		
td_te_tagdata0		
td_te_tagdata1		
td_te_tagdata2		
td_te_tagdata3		
td_te_countx		
td_te_county		
td_te_rtdtagsense		
td_te_readsremaining		
TFS (Tag Format Structure)		
tfsi_adr0[8:0]	In	Read address for dot0
tfsi_adr1[8:0]	In	Read address for dot1
Bandstore Signals		
cdu_startofbandstore[24:0]	In	Start memory area allocated for page bands
cdu_endofbandstore[24:0]	In	Last address of the memory allocated for page bands
te_finishedband	Out	Tag encoder band finished

26.7.2 Introduction

The tag data interface is responsible for obtaining the raw tag data and encoding it as required by the tag encoder. The smallest typical tag placement is 2mm × 2mm, which means a tag is at least 126 1600 dpi dots wide.

In PEC1, in order to keep up with the HCU which processes 2 dots per cycle, the tag data interface has been designed to be capable of encoding a tag in 63 cycles. This is actually accomplished in approximately 52 cycles within PEC1. For SoPEC the TE need only produce one dot per cycle; it should be able to produce tags in no more than twice the time taken by the PEC1 TE. Moreover, any change in implementation from two dots to one dot per cycle should not lose the 63/52 cycle performance edge attained in the PEC1 TE.

As shown in Figure 198, the tag data interface contains a raw tag data interface FSM that fetches tag data from DRAM, two symbol-at-a-time GF(2^4) Reed-Solomon encoders, an encoded data interface and a state machine for controlling the encoding process. It also contains a *tagData* register that needs to be set up to hold the fixed tag data for the page.

The type of encoding used depends on the registers *TE_encodefixed, TE_dataredun* and *TE_decode2den* the options being,

- (15,5) RS coding, where every 5 input symbols are used to produce 15 output symbols, so the output is 3 times the size of the input. This can be performed on fixed and variable tag data.

- (15,7) RS coding, where every 7 input symbols are used to produce 15 output symbols, so for the same number of input symbols, the output is not as large as the (15,5) code (for more details see section 26.7.6 on page 435). This can be performed on fixed and variable tag data.

- 2D decoding, where each 2 input bits are used to produce 4 output bits. This can be performed on fixed and variable tag data.

- no coding, where the data is simply passed into the Encoded Data Interface. This can be performed on fixed data only.

Each tag is made up of fixed tag data (i.e. this data is the same for each tag on the page) and variable tag data (i.e. different for each tag on the page).

Fixed tag data is either stored in DRAM as 120-bits when it is already coded (or no coding is required), 40-bits when (15,5) coding is required or 56-bits when (15,7) coding is required. Once the fixed tag data is coded it is 120-bits long. It is then stored in the Encoded Tag Data Interface. The variable tag data is stored in the DRAM in uncoded form. When (15,5) coding is required, the 120-bits stored in DRAM are encoded into 360-bits. When (15,7) coding is required, the 112-bits stored in DRAM are encoded into 240-bits. When 2D decoding is required the 120-bits stored in DRAM are converted into 240-bits. In each case the encoded bits are stored in the Encoded Tag Data Interface.

The encoded fixed and variable tag data are eventually used to print the tag.

The fixed tag data is loaded in once from the DRAM at the start of a page. It is encoded as necessary and is then stored in one of the 8x15-bits registers/RAMs in the Encoded Tag Data Interface. This data remains unchanged in the registers/RAMs until the next page is ready to be processed.

The 120-bits of unencoded variable tag data for each tag is stored in four 32-bit words. The TE re-reads the variable tag data, for a particular tag from DRAM, every time it produces that tag. The variable tag data FIFO which reads from DRAM has enough space to store 4 tags.

26.7.2.1 Bandstore wrapping

Both TD and TFS storage in DRAM can wrap around the bandstore area. The bounds of the band store are described by inputs from the CDU shown in Table . The TD and TFS DRAM interfaces therefore support bandstore wrapping. If the TD or TFS DRAM interface increments an address it is checked to see if it matches the end of bandstore address. If so, then the address is mapped to the start of the bandstore.

26.7.3   Data Flow

An overview of the dataflow through the TDI can be seen in Figure 198 below.

The TD interface consists of the following main sections:

- the Raw Tag Data Interface - fetches tag data from DRAM;
- the tag data register;

- 2 Reed Solomon encoders - each encodes one 4-bit symbol at a time;
- the Encoded Tag Data Interface - supplies encoded tag data for output;
- Two 2D decoders.

The main performance specification for PEC1 is that the TE must be able to output data at a continuous rate of 2 dots per cycle.

26.7.4   Raw tag data interface

The raw tag data interface (RTDI) provides a simple means of accessing raw tag data in DRAM. The RTDI passes tag data into a FIFO where it can be subsequently read as required. The 64-bit output from the FIFO can be read directly, with the value of the *wr_rd_counter* being used to set/reset as the enable signal (*rtdAvail*). The FIFO is clocked out with receipt of an *rtdRd* signal from the TS FSM.

Figure 199 shows a block diagram of the raw tag data interface.

26.7.4.1   RTDI FSM

The RTDI state machine is responsible for keeping the raw tag FIFO full. The state machine reads the line of tag data once for each printline that uses the tag. This means a given line of tag data will be read *TagHeight* times. Typically this will be 126 times or more, based on an approximately 2mm tag. Note that the first line of tag data may be read fewer times since the start of the page may be within a tag. In addition odd and even rows of tags may contain different numbers of tags. Section 26.6.5.1 outlines how to start the TE and restart it between bands. Users must set the *NextBandStartTagDataAdr, NextBandEndOfTagData, NextBandFirstTagLineHeight and numTags[0], numTags[1]* registers before starting the TE by asserting Go.

To restart the tag encoder for second and subsequent bands of a page, the *NextBandStartTagDataAdr, NextBandEndOfTagData* and *NextBandFirstTagLineHeight* registers need to be updated (typically *numTags[0]* and *numTags[1]* will be the same if the previous band contains an even number of tag rows) and *NextBandEnable* set. See Section 26.6.5.1 for a full description of the four ways of reprogramming the TE between bands.

The tag data is read once for every printline containing tags. When maximally packed, a row of tags contains 163 tags (see Table n page465 on page 408).

The RTDI State Flow diagram is shown in Figure 200. An explanation of the states follows:

idle state:- Stay in the idle state if there is no variable data present. If there is variable data present and there are at least 4 spaces left in the FIFO then request a burst of 2 tags from the DRAM (1 * 256bits). Counter *countx* is assigned the number of tags in a even/odd line which depends on the value of register *rtdtagsense*. Down-counter *county* is assigned the number of dot lines high a tag will be (min 126). Initially it must be set the *firsttaglineheight* value as the TE may be between pages (i.e. a partial tag). For normal tag generation *county* will take the value of *tagmaxline* register.

diu_access:- The diu_access state will generate a request to the DRAM if there are at least 4 spaces in the FIFO. This is indicated by the counter *wr_rd_counter* which is incremented/decremented on writes/reads of the FIFO. As long as *wr_rd_counter* is less than 4 (FIFO is 8 high) there must be 4 locations free. A control signal called *td_diu_radrvalid* is

432

generated for the duration of the DRAM burst access. Addresses are sent in bursts of 1. The counter *burst_count* controls this signal, (will involve modification to existing TE code.)

If there is an odd number of tags in line then the last DRAM read will contain a tag in the first 128 bits and padding in the final 128 bits.

5　　*fifo_load*:- This state controls the addressing to the DRAM. Counters *countx* and *county* are used to monitor whether the TE is processing a line of dots within a row of tags. When *countx* is zero it means all tag dots for this row are complete. When *county* is zero it means the TE is on the last line of dots (prior to Y scaling) for this row of tags. When a row of tags is complete the sense of *rtdtagsense* is inverted (odd/ even). The *rawtagdataadr* is compared to the *te_endoftagdata*

10　　address. If *rawtagdataadr = endoftagdata* the *doneband* signal is set, the *finishedband* signal is pulsed, and the FSM enters the *rtd_stall* state until the *doneband* signal is reset to zero by the PCU by which time the *rawtagdata*, *endoftagedata* and *firsttaglineheight* registers are setup with new values to restart the TE. This state is used to count the 64-bit reads from the DIU. Each time *diu_td_rvalid* is high *rtd_data_count* is incremented by 1. The compare of *rtd_data_count* =

15　　*rtd_num* is necessary to find out when either all 4*64-bit data has been received or n*64-bit data (depending on a match of *rawtagdataadr = endoftagdata* in the middle of a set of 4*64-bit values being returned by the DIU.

rtd_stall:- This state waits for the the *doneband* signal to be reset (see page 426 for a description of how this occurs). Once reset the FSM returns to the idle state. This states also performs the

20　　same count on the *diu_data* read as above in the case where *diu_td_rvalid* has not gone high by the time the addressing is complete and the end of band data has been reached i.e.

rawtagdataadr = endoftagdata

26.7.5　　TDI state machine

The tag data state machine has two processing phases. The first processing phase is to encode

25　　the fixed tag data stored in the 128-bit (2 × 64-bit) tag data register. The second is to encode tag data as it is required by the tag encoder.

When the Tag Encoder is started up, the fixed tag data is already preloaded in the 128 bit tag data record. If *encodeFixed* is set, then the 2 codewords stored in the lower bits of the tag data record need to be encoded: 40 bits if *dataRedun* = 0, and 56 bits if *dataRedun* = 1. If *encodeFixed*

30　　is clear, then the lower 120 bits of the tag data record must be passed to the encoded tag data interface without being encoded.

When *encodeFixed* is set, the symbols derived from codeword 0 are written to codeword 6 and the symbols derived from codeword 1 are written to codeword 7. The data symbols are stored first and then the remaining redundancy symbols are stored afterwards, for a total of 15 symbols.

35　　Thus, when *dataRedun* = 0, the 5 symbols derived from bits 0-19 are written to symbols 0-4, and the redundancy symbols are written to symbols 5-14. When *dataRedun* = 1, the 7 symbols derived from bits 0-27 are written to symbols 0-6, and the redundancy symbols are written to symbols 7-14.

When *encodeFixed* is clear, the 120 bits of fixed data is copied directly to codewords 6 and 7.

40　　The TDI State Flow diagram is shown in Figure 202. An explanation of the states follows.

idle:- In the idle state wait for the tag encoder *go* signal - *top_go* = 1. The first task is to either store or encode the Fixed data. Once the Fixed data is stored or encoded/stored the *donefixed* flag is set. If there is no variable data the FSM returns to the idle state hence the reason to check the *donefixed* flag before advancing i.e. only store/encode the fixed data once.

5   *fixed_data:-* In the fixed_data state the FSM must decode whether to directly store the fixed data in the ETDi or if the fixed data needs to be either (15:5) (40-bits) or (15:7) (56-bits) RS encoded or 2D decoded. The values stored in registers *encodefixed* and *dataredun* and *decode2den* determine what the next state should be.

bypass_to_etdi:- The bypass_to_etdi takes 120-bits of fixed data(pre-encoded) from the

10   *tag_data(127:0)* register and stores it in the 15*8 (by 2 for simultaneous reads) buffers. The data is passed from the *tag_data* register through 3 levels of muxing (level1, level2, level3) where it enters the RS0/RS1 encoders (which are now in a straight through mode (i.e. *control_5* and *control_7* are zero hence the data passes straight from the input to the output). The MSBs of the *etd_wr_adr* must be high to store this data as codewords 6,7.

15   *etd_buf_switch:-* This state is used to set the *tdvalid* signal and pulse the *etd_adv_tag* signal which in turn is used to switch the read write sense of the ETDi buffers (*wrsb0*). The *firsttime* signal is used to identify the first time a tag is encoded. If zero it means read the tag data from the RTDi FIFO and encode. Once encoded and stored the FSM returns to this state where it evaluates the sense of *tdvalid*. First time around it will be zero so this sets *tdvalid* and returns to

20   the readtagdata state to fill the 2nd ETDi buffer. After this the FSM returns to this state and waits for the *lastdotintag* signal to arrive. In between tags when the lastdotingtag signal is received the *etd_adv_tag* is pulsed and the FSM goes to the readtagdata state. However if the *lastdotintag* signal arrives at the end of a line there is an extra 1 cycle delay introduced in generating the *etd_adv_tag* pulse (via *etd_adv_tag_endofline*) due to the pipelining in the TFS. This allows all the

·25   previous tag to be read from the correct buffer and seamless transfer to the other buffer for the next line.

readtagdata:- The readtagdata state waits to receive a *rtdavail* signal from the raw tag data interface which indicates there is raw tag data available. The *tag_data* register is 128-bits so it takes 2 pulses of the *rtdrd* signal to get the 2*64-bits into the *tag_data* register. If the *rtdavail*

30   signal is set *rtdrd* is pulsed for 1 cycle and the FSM steps onto the loadtagdata state. Initially the flag *first64bits* will be zero. The 64-bits of *rtd* are assigned to the *tag_data[63:0]* and the flag *first64bits* is set to indicate the first raw tag data read is complete. The FSM then steps back to the read_tagdata state where it generates the second *rtdrd* pulse. The FSM then steps onto the loadtagdata state for where the second 64-bits of rawtag data are assigned to *tag_data[128:64]*.

35   *loadtagdata:-* The loadtagdata state writes the raw tag data into the *tag_data* register from the RTDi FIFO. The *first64bits* flag is reset to zero as the *tag_data* register now contains 120/112 bits of variable data. A decode of whether to (15:5) or (15:7) RS encode or 2D decode this data decides the next state.

rs_15_5:- The rs_15_5 (Reed Solomon (15:5) mode) state either encodes 40-bit Fixed data or

40   120-bit Variable data and provides the encoded tag data write address and write enable

(*etd_wr_adr* and *etdwe* respectively). Once the fixed tag data is encoded the *donefixed* flag is set as this only needs to be done once per page. The *variabledatapresent* register is then polled to see if there is variable data in the tags. If there is variable data present then this data must be read from the RTDi and loaded into the *tag_data* register. Else the *tdvalid* flag must be set and FSM returns to the idle state. control_5 is a control bit for the RS Encoder and controls feedforward and feedback muxes that enable (15:5) encoding.

The rs_15_5 state also generates the control signals for passing 120-bits of variable tag data to the RS encoder in 4-bit symbols per clock cycle. *rs_counter* is used both to control the level1_mux and act as the 15-cycle counter of the RS Encoder. This logic cycles for a total of 3*15 cycles to encode the 120-bits.

rs_15_7:- The rs_15_7 state is similar to the rs_15_5 state except the level1_mux has to select 7 4-bit symbols instead of 5.

decode_2d_15_5, decode_2d_15_7:- The decode_2d states provides the control signals for passing the 120-bit variable data to the 2D decoder. The 2 lsbs are decoded to create 4 bits. The 4 bits from each decoder are combined and stored in the ETDi. Next the 2 MSBs are decoded to create 4 bits. Again the 4 bits from each decoder are combined and stored in the ETDi.

As can be seen from Figure n page488 on page **Error! Bookmark not defined.** there are 3 stages of muxing between the Tag Data register and the RS encoders or 2D decoders. Levels 1-2 are controlled by *level1_mux* and *level2_mux* which are generated within the TDi FSM as is the write address to the ETDi buffers (*etd_wr_adr*)

Figures 203 through 208 illustrate the mappings used to store the encoded fixed and variable tag data in the ETDI buffers.

26.7.6   Reed Solomon (RS) Encoder

26.7.7   Introduction

A Reed Solomon code is a non binary, block code. If a symbol consists of m bits then there are q = 2^m possible symbols defining the code alphabet. In the TE, m = 4 so the number of possible symbols is q = 16.

An (n,k) RS code is a block code with k information symbols and n code-word symbols. RS codes have the property that the code word n is limited to at most q+1 symbols in length.

In the case of the TE, both (15,5) and (15,7) RS codes can be used. This means that up to 5 and 4 symbols respectively can be corrected.

Only one type of RS coder is used at any particular time. The RS coder to be used is determined by the registers *TE_dataredun* and *TE_decode2den*:

- *TE_dataredun* = 0 and *TE_decode2den* = 0, then use the (15,5) RS coder
- *TE_dataredun* = 1 and *TE_decode2den* = 0, then use the (15,7) RS coder

For a (15,k) RS code with m = 4, k 4-bit information symbols applied to the coder produce 15 4-bit codeword symbols at the output. In the TE, the code is systematic so the first k codeword symbols are the same the as the k input information symbols.

A simple block diagram can be seen in.

26.7.8   I/O Specification

A I/O diagram of the RS encoder can be seen in.

26.7.9   Proposed implementation

In the case of the TE, (15,5) and (15,7) codes are to be used with 4-bits per symbol.

The primitive polynomial is $p(x) = x^4 + x + 1$

5   In the case of the (15,5) code, this gives a generator polynomial of

$$g(x) = (x+a)(x+a^2)(x+a^3)(x+a^4)(x+a^5)(x+a^6)(x+a^7)(x+a^8)(x+a^9)(x+a^{10})$$

$$g(x) = x^{10} + a^2x^9 + a^3x^8 + a^9x^7 + a^6x^6 + a^{14}x^5 + a^2x^4 + ax^3 + a^6x^2 + ax + a^{10}$$

$$g(x) = x^{10} + g_9x^9 + g_8x^8 + g_7x^7 + g_6x^6 + g_5x^5 + g_4x^4 + g_3x^3 + g_2x^2 + g_1x + g_0$$

10

In the case of the (15,7) code, this gives a generator polynomial of

$$h(x) = (x+a)(x+a^2)(x+a^3)(x+a^4)(x+a^5)(x+a^6)(x+a^7)(x+a^8)$$

$$h(x) = x^8 + a^{14}x^7 + a^2x^6 + a^4x^5 + a^2x^4 + a^{13}x^3 + a^5x^2 + a^{11}x + a^6$$

$$h(x) = x^8 + h_7x^7 + h_6x^6 + h_5x^5 + h_4x^4 + h_3x^3 + h_2x^2 + h_1x + h_0$$

15   The output code words are produced by dividing the generator polynomial into a polynomial made up from the input symbols.

This division is accomplished using the circuit shown in Figure 211.

The data in the circuit are Galois Field elements so addition and multiplication are performed using special circuitry. These are explained in the next sections.

20   The RS coder can operate either in (15,5) or (15,7) mode. The selection is made by the registers *TE_dataredun* and *TE_decode2den*.

When operating in (15,5) mode *control_7* is always zero and when operating in (15,7) mode *control_5* is always zero.

Firstly consider (15,5) mode i.e. *TE_dataredun* is set to zero.

25   For each new set of 5 input symbols, processing is as follows:

The 4-bits of the first symbol d_0 are fed to the input port *rs_data_in(3:0)* and *control_5* is set to 0. *mux2* is set so as to use the output as feedback. *control_5* is zero so *mux4* selects the input (*rs_data_in*) as the output (*rs_data_out*). Once the data has settled (<< 1 cycle), the shift registers are clocked. The next symbol d_1 is then applied to the input, and again after the data has settled

30   the shift registers are clocked again. This is repeated for the next 3 symbols d_2, d_3 and d_4. As a result, the first 5 outputs are the same as the inputs. After 5 cycles, the shift registers now contain the next 10 required outputs. *control_5* is set to 1 for the next 10 cycles so that zeros are fed back by *mux2* and the shift register values are fed to the output by *mux3* and *mux4* by simply clocking the registers.

35   A timing diagram is shown below.

Secondly consider (15,7) mode i.e. *TE_dataredun* is set to one.

In this case processing is similar to above except that *control_7* stays low while 7 symbols (d_0, d_1 ... d_6) are fed in. As well as being fed back into the circuit, these symbols are fed to the output. After these 7 cycles, *control_7* is set to 1 and the contents of the shift registers are fed to the

40   output.

A timing diagram is shown below.

The *enable* signal can be used to start/reset the counter and the shift registers.

The RS encoders can be designed so that encoding starts on a rising *enable* edge. After 15 symbols have been output, the encoder stops until a rising *enable* edge is detected. As a result

5   there will be a delay between each codeword.

Alternatively, once the enable goes high the shift registers are reset and encoding will proceed until it is told to stop. *rs_data_in* must be supplied at the correct time. Using this method, data can be continuously output at a rate of 1 symbol per cycle, even over a few codewords.

Alternatively, the RS encoder can request data as it requires.

10   The performance criterion that must be met is that the following must be carried out within 63 cycles

- load one tag's raw data into *TE_tagdata*
- encode the raw tag data
- store the encoded tag data in the Encoded Tag Data Interface

15   In the case of the raw fixed tag data at the start of a page, there is no definite performance criterion except that it should be encoded and stored as fast as possible.

26.7.10   Galois Field elements and their representation

A Galois Field is a set of elements in which we can do addition, subtraction, multiplication and division without leaving the set.

20   The TE uses RS encoding over the Galois Field $GF(2^4)$. There are 2^4 elements in $GF(2^4)$ and they are generated using the primitive polynomial $p(x) = x^4 + x + 1$.

The 16 elements of $GF(2^4)$ can be represented in a number of different ways. Table 179 shows three possible representations - the power, polynomial and 4-tuple representation.

Table 179. $GF(2^4)$ representations

25

power representation	Polynomial Representation	4-tuple representation (a0 a1 a2 a3)
0	0	(0 0 0 0)
1	1	(1 0 0 0)
A	x	(0 1 0 0)
α^2	x^2	(0 0 1 0)
α^3	x^3	(0 0 0 1)
α^4	$1 + x$	(1 1 0 0)
α^5	$x + x^2$	(0 1 1 0)
a^6	$x^2 + x^3$	(0 0 1 1)
α^7	$1 + x + x^3$	(1 1 0 1)
α^8	$1 + x^2$	(1 0 1 0)
α^9		(0 1 0 1)

	x ≤≤≤≤≤≤≤≤≤+ x³	
α^{10}	$1 + x + x^2$	(1 1 1 0)
α^{11}	$x + x^2 + x^3$	(0 1 1 1)
α^{12}	$1 + x + x^2 + x^3$	(1 1 1 1)
α^{13}	$1 \quad + x^2 + x^3$	(1 0 1 1)
α^{14}	$1 \quad\quad + x^3$	(1 0 0 1)

26.7.11 Multiplication of GF(2^4) elements

The multiplication of two field elements α^a and α^b is defined as

$$\alpha^c = \alpha^a . \alpha^b = \alpha^{(a+b)\text{modulo } 15}$$

Thus

$$\alpha^1 . \alpha^2 = \alpha^3$$
$$\alpha^5 . \alpha^{10} = \alpha^{15}$$
$$\alpha^6 . \alpha^{12} = \alpha^3$$

So if we have the elements in exponential form, multiplication is simply a matter of modulo 15 addition.

If the elements are in polynomial/tuple form, the polynomials must be multiplied and reduced mod $x^4 + x + 1$.

Suppose we wish to multiply the two field elements in GF(2^4):

$$\alpha^a = a_3 x^3 + a_2 x^2 + a_1 x^1 + a_0$$
$$\alpha^b = b_3 x^3 + b_2 x^2 + b_1 x^1 + b_0$$

where a_i, b_i are in the field (0,1) (i.e. modulo 2 arithmetic)

Multiplying these out and using $x^4 + x + 1 = 0$ we get:

$$\alpha^{a+b} = [(a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0) + a_3 b_3] x^3$$
$$+ [(a_0 b_2 + a_1 b_1 + a_2 b_0) + a_3 b_3 + (a_3 b_2 + a_2 b_3)] x^2$$
$$+ [(a_0 b_1 + a_1 b_0) + (a_3 b_2 + a_2 b_3) + (a_1 b_3 + a_2 b_2 + a_3 b_1)] x$$
$$+ [(a_0 b_0 + a_1 b_3 + a_2 b_2 + a_3 b_1)]$$

$$\alpha^{a+b} = [a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 (b_0 + b_3)] x^3$$
$$+ [a_0 b_2 + a_1 b_1 + a_2 (b_0 + b_3) + a_3 (b_2 + b_3)] x^2$$
$$+ [a_0 b_1 + a_1 (b_0 + b_3) + a_2 (b_2 + b_3) + a_3 (b_1 + b_2)] x$$
$$+ [a_0 b_0 + a_1 b_3 + a_2 b_2 + a_3 b_1]$$

If we wish to multiply an arbitrary field element by a fixed field element we get a more simple form. Suppose we wish to multiply α^b by α^3.

In this case $\alpha^3 = x^3$ so (a0 a1 a2 a3) = (0 0 0 1). Substituting this into the above equation gives

$$\alpha^c = (b_0 + b_3) x^3 + (b_2 + b_3) x^2 + (b_1 + b_2) x + b_1$$

This can be implemented using simple XOR gates as shown in Figure 214

26.7.12 Addition of GF(2^4) elements

If the elements are in their polynomial/tuple form, polynomials are simply added.

Suppose we wish to add the two field elements in GF(2^4):

$$\alpha^a = a_3x^3 + a_2x^2 + a_1x + a_0$$

$$\alpha^b = b_3x^3 + b_2x^2 + b_1x + b_0$$

where a_i, b_i are in the field (0,1) (i.e. modulo 2 arithmetic)

$$\alpha^c = \alpha^a + \alpha^b = (a_3 + b_3)x^3 + (a_2 + b_2)x^2 + (a_1 + b_1)x + (a_0 + b_0)$$

Again this can be implemented using simple XOR gates as shown in Figure 215

26.7.13   Reed Solomon Implementation

The designer can decide to create the relevant addition and multiplication circuits and instantiate them where necessary. Alternatively the feedback multiplications can be combined as follows. Consider the multiplication

$$\alpha^a . \alpha^b = \alpha^c$$

or in terms of polynomials

$$(a_3x^3 + a_2x^2 + a_1x + a_0).(b_3x^3 + b_2x^2 + b_1x + b_0) = (c_3x^3 + c_2x^2 + c_1x + c_0)$$

If we substitute all of the possible field elements in for α^a and express α^c in terms of α^b, we get the table of results shown in Table 180.

Table 180. α^c multiplied by all field elements, expressed in terms of α^b

$\alpha a = a3x3 + a2x2 + a1x + a0$		$c3x3 + c2x2 + c1x + c0$			
fixed field element	(a0 a1 a2 a3)	c0	c1	c2	c3
0	(0 0 0 0)				
1	(1 0 0 0)	b_0	b_1	b_2	b_3
a	(0 1 0 0)	b_3	b_0+b_3	b_1	b_2
α^2	(0 0 1 0)	b_2	b_2+b_3	b_0+b_3	b_1
α^3	(0 0 0 1)	b_1	b_1+b_2	b_2+b_3	b_0+b_3
α^4	(1 1 0 0)	b_0+b_3	$b_0+b_1+b_3$	b_1+b_2	b_2+b_3
α^5	(0 1 1 0)	b_2+b_3	b_0+b_2	$b_0+b_1+b_3$	b_1+b_2
a^6	(0 0 1 1)	b_1+b_2	b_1+b_3	b_0+b_2	$b_0+b_1+b_3$
α^7	(1 1 0 1)	$b_0+b_1+b_3$	$b_0+b_2+b_3$	b_1+b_3	b_0+b_2
α^8	(1 0 1 0)	b_0+b_2	$b_1+b_2+b_3$	$b_0+b_2+b_3$	b_1+b_3
α^9	(0 1 0 1)	b_1+b_3	$b_0+b_1+b_2+b_3$	$b_1+b_2+b_3$	$b_0+b_2+b_3$
α^{10}	(1 1 1 0)	$b_0+b_2+b_3$	$b_0+b_1+b_2$	$b_0+b_1+b_2+b_3$	$b_1+b_2+b_3$
α^{11}	(0 1 1 1)	$b_1+b_2+b_3$	b_0+b_1	$b_0+b_1+b_2$	$b_0+b_1+b_2+b_3$
α^{12}	(1 1 1 1)	$b_0+b_1+b_2+b_3$	b_0	b_0+b_1	$b_0+b_1+b_2$
α^{13}	(1 0 1 1)	$b_0+b_1+b_2$	b_3	b_0	b_0+b_1
α^{14}	(1 0 0 1)	b_0+b_1	b_2	b_3	b_0

the following signals are required:

- b_0, b_1, b_2, b_3,
- (b_0+b_1), (b_0+b_2), (b_0+b_3), (b_1+b_2), (b_1+b_3), (b_2+b_3),
- $(b_0+b_1+b_2)$, $(b_0+b_1+b_3)$, $(b_0+b_2+b_3)$, $(b_1+b_2+b_3)$,
- $(b_0+b_1+b_2+b_3)$

The implementation of the circuit can be seen in Figure . The main components are XOR gates, 4-bit shift registers and multiplexers.

The RS encoder has 4 input lines labelled 0,1,2 & 3 and 4 output lines labelled 0,1,2 & 3. This labelling corresponds to the subscripts of the polynomial/4-tuple representation. The mapping of 4-bit symbols from the TE_tagdata register into the RS is as follows:

- the LSB in the TE_tagdata is fed into line0
- the next most significant LSB is fed into line1
- the next most significant LSB is fed into line2
- the MSB is fed into line3

The RS output mapping to the Encoded tag data interface is similiar. Two encoded symbols are stored in an 8-bit address. Within these 8 bits:

- line0 is fed into the LSB (bit 0/4)
- line1 is fed into the next most significant LSB (bit 1/5)
- line2 is fed into the next most significant LSB (bit 2/6)
- line3 is fed into the MSB (bit 3/7)

267.14   2D Decoder

The 2D decoder is selected when TE_decode2den = 1. It operates on variable tag data only. its function is to convert 2-bits into 4-bits according to Table 181..

Table 181. Operation of 2D decoder

input	output
0 0	0 0 0 1
0 1	0 0 1 0
1 0	0 1 0 0
1 1	1 0 0 0

26.7.15 Encoded tag data interface

The encoded tag data interface contains an encoded fixed tag data store interface and an encoded variable tag data store interface, as shown in Figure 217.

The two reord units simply reorder the 9 input bits to map low-order codewords into the bit selection component of the address as shown in Table 182. Reordering of write addresses is not necessary since the addresses are already in the correct format.

Table 182. Reord unit

bit#	input bit	interpretation	output bit	interpretation
8	A	select 1 of 8 codewords	A	select 1 of 4 codeword tables
7	B		B	
6	C		D	select 1 of 15 symbols
5	D	select 1 of 15 symbols	E	
4	E		F	
3	F		G	
2	G		C	select 1 of 8 bits
1	H	select 1 of 4 bits	H	
0	I		I	

The encoded fixed data interface is a single 15 × 8-bit RAM with 2 read ports and 1 write port. As it is only written to during page setup time (it is fixed for the duration of a page) there is no need for simultaneous read/write access. However the fixed data store must be capable of decoding two simultaneous reads in a single cycle.Figure 218 shows the implementation of the fixed data store.

The encoded variable tag data interface is a double buffered 3 × 15 × 8-bit RAM with 2 read ports and 1 write port. The double buffering allows one tag's data to be read (two reads in a single cycle) while the next tag's variable data is being stored. Write addressing is 6 bits: 2 bits of address for selecting 1 of 3, and 4 bits of address for selecting 1 of 15. Read addressing is the same with the addition of 3 more address bits for selecting 1 of 8.

Figure 219 shows the implementation of the encoded variable tag data store. Double buffering is implemented via two sub-buffers. Each time an *AdvTag* pulse is received, the sense of which sub-buffer is being read from or written to changes. This is accomplished by a 1-bit flag called *wrsb0*. Although the initial state of *wrsb0* is irrelevant, it must invert upon receipt of an *AdvTag* pulse. The structure of each sub-buffer is shown in Figure 220.

26.8 TAG FORMAT STRUCTURE (TFS) INTERFACE

26.8.1 Introduction

The TFS specifies the contents of every dot position within a tags border i.e.:

- is the dot part of the background?
- is the dot part of the data?

The TFS is broken up into Tag Line Structures (TLS) which specify the contents of every dot position in a particular line of a tag. Each TLS consists of three tables - A, B and C (see Figure 221).

For a given line of dots, all the tags on that line correspond to the same tag line structure. Consequently, for a given line of output dots, a single tag line structure is required, and not the entire TFS. Double buffering allows the next tag line structure to be fetched from the TFS in DRAM while the existing tag line structure is used to render the current tag line.

The TFS interface is responsible for loading the appropriate line of the tag format structure as the tag encoder advances through the page. It is also responsible for producing table A and table B outputs for two consecutive dot positions in the current tag line.

- There is a TLS for every dot line of a tag.
- All tags that are on the same line have the exact same TLS.
- A tag can be up to 384 dots wide, so each of these 384 dots must be specified in the TLS.
- The TLS information is stored in DRAM and one TLS must be read into the TFS Interface for each line of dots that are outputted to the Tag Plane Line Buffers.
- Each TLS is read from DRAM as 5 times 256-bit words with 214 padded bits in the last 256-bit DRAM read.

26.8.2   I/O Specification

Table 183. Tag Format Structure Interface Port List

signal name	signal type	description
Pclk	In	SoPEC system clock
prst_n	In	Active-low, synchronous reset in pclk domain
top_go	In	Go signal from TE top level
DRAM		
diu_data[63:0]	In	Data from DRAM
diu_tfs_rack	In	Data acknowledge from DRAM
diu_tfs_rvalid	In	Data valid from DRAM
tfs_diu_rreq	Out	Read request to DRAM
tfs_diu_radr[21:5]	Out	Read address to DRAM
tag encoder top level		
top_advtagline	In	Pulsed after the last line of a row of tags
top_tagaltsense	In	For even tag rows = 0 i.e. 0,2,4.. For odd tag rows = 1 i.e. 1,3,5...
top_lastdotintag	In	Last dot in tag is currently being processed
top_dotposvalid	In	Current dot position is a tag dot and its structure data and tag data is available
top_tagdotnum[7:0]	In	Counts from zero up to *TE_tagmaxdotpairs* (min. =1, max. = 192)
tfsi_valid	Out	TLS tables A, B and C, ready for use
tfsi_ta_dot0[1:0]	Out	Even entry from Table A corresponding to top_tagdotnum
tfsi_ta_dot1[1:0]	Out	Odd entry from Table A corresponding to top_tagdotnum
tag encoder top level (PCU read decoder)		

442

tfs_te_tfsstartadr[23:0]	Out	TFS tfsstartadr register	
tfs_te_tfsendadr[23:0]	Out	TFS tfsendadr register	
tfs_te_tfsfirstlineadr[23:0]	Out	TFS tfsfirstlineadr register	
tfs_te_currtfsadr[23:0]	Out	TFS currtfsadr register	
TDI			
tfsi_tdi_adr0[8:0]	Out	Read address for dot0 (even dot)	
tfsi_tdi_adr1[8:0]	Out	Read address for dot1 (odd dot)	

26.8.2.1 State machine

The state machine is responsible for generating control signals for the various TFS table units, and to load the appropriate line from the TFS. The states are explained below.

idle:- Wait for *top_go* to become active. Pulse *adv_tfs_line* for 1 cycle to reset *tawradr* and *tbwradr*

5   registers. Pulsing *adv_tfs_line* will switch the read/write sense of Table B so switching Table A here as well to keep things the same i.e. *wrta0* = NOT(*wrta0*).

diu_access:- In the diu_access state a request is sent to the DIU. Once an *ack* signal is received Table A write enable is asserted and the FSM moves to the *tls_load* state.

tls_load:- The DRAM access is a burst of 5 256-bit accesses, ultimately returned by the DIU as

10   5*(4*64bit) words. There will be 192 padded bits in the last 256-bit DRAM word. The first 12 64-bit words reads are for Table A, words 12 to 15 and some of 16 are for Table B while part of read 16 data is for Table C. The counter *read_num* is used to identify which data goes to which table. The table B data is stored temporarily in a 288-bit register until the tls_update state hence *tbwe* does not become active until read_num = 16).

15   •   The DIU data goes directly into Table A (12 * 64).

•   The DIU data for Table B is loaded into a 288-bit register.

•   The DIU data goes directly into Table C.

tls_update:- The 288-bits in Table B need to written to a 32*9 buffer. The *tls_update* state takes

20   care of this using the *read_num* counter.

tls_next:- This state checks the logic level of *tfsvalid* and switches the read/write senses of Table A (*wrta0*) and Table B a cycle later (using the *adv_tfs_line* pulse). The reason for switching Table A a cycle early is to make sure the top_level address via *tagdotnum* is pointing to the correct buffer. Keep in mind the top_level is working a cycle ahead of Table A and 2 cycles ahead of

25   Table B.

If *tfsValid* is 1, the state machine waits until the *advTagLine* signal is received. When it is received, the state machine pulses *advTFSLine* (to switch read/write sense in tables A, B, C), and starts reading the next line of the TFS from *currTFSAdr*.

If *tfsValid* is 0, the state machine pulses *advTFSLine* (to switch read/write sense in tables A, B, C)

30   and then jumps to the tls_tfsvalid_set state where the signal *tfsValid* is set to 1 (allowing the tag encoder to start, or to continue if it had been stalled). The state machine can then start reading the next line of the TFS from *currTFSAdr*.

tls_tfsvalid_next:- Simply sets the *tfsvalid* signal and returns the FSM to the diu_access state.

If an *advTagLine* signal is received before the next line of the TFS has been read in, *tfsValid* is cleared to 0 and processing continues as outlined above.

26.8.2.2 Bandstore wrapping

Both TD and TFS storage in DRAM can wrap around the bandstore area. The bounds of the band store are described by inputs from the CDU shown in Table . The TD and TFS DRAM interfaces therefore support bandstore wrapping. If the TD or TFS DRAM interface increments an address it is checked to see if it matches the end of bandstore address. If so, then the address is mapped to the start of the bandstore.

The TFS state flow diagram is shown in below.

26.8.3 Generating a tag from Tables A, B and C

The TFS contains an entry for each dot position within the tag's bounding box. Each entry specifies whether the dot is part of the constant background pattern or part of the tag's data component (both fixed and variable).

The TFS therefore has TagHeight x TagWidth entries, where TagHeight is the height of the tag in dot-lines and TagWidth is the width of the tag in dots. The TFS entries that specify a single dot-line of a tag are known as a Tag Line Structure.

The TFS contains a TLS for each of the 1600 dpi lines in the tag's bounding box. Each TLS contains three contiguous tables, known as tables A, B and C.

Table A contains 384 2-bit entries i.e. one entry for each dot in a single line of a tag up to the maximum width of a tag. The actual number of entries used should match the size of the bounding box for the tag in the dot dimension, but all 384 entries must be present.

Table B contains 32 9-bit data address that refer to (in order of appearance) the data dots present in the particular line. Again, all 32 entries must be present, even if fewer are used.

Table C contains two 5-bit pointers into table B and is followed by 22 unused bits. The total length of each TLS is therefore 34 32-bit words.

Each output dot value is generated as follows: Each entry in Table A consists of 2-bits - bit0 and bit1. These 2-bits are interpreted according to Table 184, Table 185 and Table 186.

Table 184. Interpretation of bit0 from entry in Table A

bit0	interpretation
0	the output bit comes directly from bit1 (see Table).
1	the output bit comes from a data bit. Bit1 is used in conjunction with Tag Line Structure Table B to determine which data bit will be output.

Table 185. Interpretation of bit1 from entry in table A when bit0 = 0

bit 1	interpretation
0	output 0
1	output 1

Table 186. Interpretation of bit1 from entry in table A when bit0 = 1

bit 1	interpretation
0	output data bit pointed to by current index into Table B.
1	output data bit pointed to by current index into Table B, and advance index by 1.

If bit0 = 0 then the output dot for this entry is part of the constant background pattern. The dot value itself comes from bit1 i.e. if bit1 = 0 then the output is 0 and if bit1 = 1 then the output is 1.

If bit0 = 1 then the output dot for this entry comes from the variable or fixed tag data. Bit1 is used

5   in conjunction with Tables B and C to determine data bits to use.

To understand the interpretation of bit1 when bit0 = 1 we need to know what is stored in Table B. Table B contains the addresses of all the data bits that are used in the particular line of a tag in order of appearance. Therefore, up to 32 different data bits can appear in a line of a tag. The address of the first data dot in a tag will be given by the address stored in entry 0 of Table B. As

10   we advance along the various data dots we will advance through the various Table B entries.

Each Table B entry is 9-bits long and each points to a specific variable or fixed data bit for the tag. Each tag contains a maximum of 120 fixed and 360 variable data bits, for a total of 480 data bits. To aid address decoding, the addresses are based on the RS encoded tag data. Table   lists the interpretation of the 9-bit addresses.

15   Table 187. Interpretation of 9-bit tag data address in Table B

bit pos	name	description
8	CodeWordSelect	Select 1 of 8 codewords. Codewords 0, 1, 2, 3, 4, 5 are variable data. Codewords 6, 7 are fixed data.
7		
6		
5	SymbolSelect	Select 1 of 15 symbols (1111 invalid)
4		
3		
2		
1	BitSelect	Select 1 of 4 bits from the selected symbols
0		

If the fixed data is supplied to the TE in an unencoded form, the symbols derived from codeword 0 of fixed data are written to codeword 6 and the symbols derived from fixed data codeword 1 are written to codeword 7. The data symbols are stored first and then the remaining redundancy

20   symbols are stored afterwards, for a total of 15 symbols. Thus, when 5 data symbols are used, the 5 symbols derived from bits 0-19 are written to symbols 0-4, and the redundancy symbols are written to symbols 5-14. When 7 data symbols are used, the 7 symbols derived from bits 0-27 are written to symbols 0-6, and the redundancy symbols are written to symbols 7-14

However, if the fixed data is supplied to the TE in a pre-encoded form, the encoding could theoretically be anything. Consequently the 120 bits of fixed data is copied to codewords 6 and 7 as shown in Table 188.

Table 188. Mapping of fixed data to codeword/symbols when no redundancy encoding

input bits	output symbol range	output codeword
0-19	0-4	6
20-39	0-4	7
40-59	5-9	6
60-79	5-9	7
80-99	10-14	6
100-119	10-14	7

It is important to note that the interpretation of bit1 from Table A (when bit0 = 1) is relative. A 5-bit index is used to cycle through the data address in Table B. Since the first tag on a particular line may or may not start at the first dot in the tag, an initial value for the index into Table B is needed. Subsequent tags on the same line will always start with an index of 0, and any partial tag at the end of a line will simply finish before the entire tag has been rendered. The initial index required due to the rendering of a partial tag at the start of a line is supplied by Table C. The initial index will be different for each TLS and there are two possible initial indexes since there are effectively two types of rows of tags in terms of initial offsets.

Table C provides the appropriate start index into Table B (2 5-bit indices). When rendering even rows of tags, entry 0 is used as the initial index into Table B, and when rendering odd rows of tags, entry 1 is used as the initial index into Table B. The second and subsequent tags start at the left most dots position within the tag, so can use an initial index of 0.

26.8.4   Architecture

A block diagram of the Tag Format Structure Interface can be seen in Figure 223.

26.8.4.1   Table A interface

The implementation of table A is two 16 × 64-bit RAMs with a small amount of control logic, as shown in Figure 224. While one RAM is read from for the current line's table A data (4 bits representing 2 contiguous table A entries), the other RAM is being written to with the next line's table A data (64-bits at a time).

Note:- The Table A data to be printed (if each LSB = 0) must be passed to the top_level 2 cycles after the read of Table A due to the 2-stage pipelining in the TFS from registering Table A and Table B outputs hence this extra registering stage for the generation of ta_dot0_1cyclelater and ta_dot1_1cyclelater.

Each time an AdvTFSLine pulse is received, the sense of which RAM is being read from or written to changes. This is accomplished by a 1-bit flag called wrta0. Although the initial state of wrta0 is irrelevant, it must invert upon receipt of an AdvTFSLine pulse. A 4-bit counter called taWrAdr keeps the write address for the 12 writes that occur after the start of each line (specified by the

AdvTFSLine control input). The *tawe* (table A write enable) input is set whenever the data in is to be written to table A. The *taWrAdr* address counter automatically increments with each write to table A. Address generation for *tawe* and *taWrAdr* is shown in Table 189.

26.8.4.2 Table C interface

A block diagram of the table C interface is shown below in Figure 226.

The address generator for table C contains a 5 bit address register *adr* that is set to a new address at the start of processing the tag (either of the two table C initial values based on *tagAltSense* at the start of the line, and 0 for subsequent tags on the same line). Each cycle two addresses into table B are generated based on the two 2-bit inputs (*in0* and *in1*). As shown in Section 189, the output address *tbRdAdr0* is always *adr* and *tbRdAdr1* is one of *adr* and *adr+1*, and at the end of the cycle *adr* takes on one of *adr*, *adr+1*, and *adr+2*.

Table 189. AdrGen lookup table

inputs		outputs		
in0	in1	adr0Sel	adr1Sel	adrSel
00	00	X[18]	X	adr
00	01	X	adr	adr
00	10	X	X	adr
00	11	X	adr	adr+1
01	00	adr	X	adr
01	01	adr	adr	adr
01	10	adr	X	adr
01	11	adr	adr	adr+1
10	00	X	X	adr
10	01	X	adr	adr
10	10	X	X	adr
10	11	X	adr	adr+1
11	00	adr	X	adr+1
11	01	adr	adr+1	adr+1
11	10	adr	X	adr+1
11	11	adr	adr+1	adr+2

26.8.4.3 Table B interface

The table B interface implementation generates two encoded tag data addresses (*tfsi_adr0*, *tfsi_adr1*) based on two table B input addresses (*tbRdAdr0*, *tbRdAdr1*). A block diagram of table B can be seen in Figure 227.

---

[18]   X = don't care state.

447

Table B data is initially loaded into the 288-bit table B temporary register via the TFS FSM. Once all 288-bit entries have been loaded from DRAM, the data is written in 9-bit chunks to the 32*9 register arrays based on tbwradr.

Each time an *AdvTFSLine* pulse is received, the sense of which sub buffer is being read from or written to changes. This is accomplished by a 1-bit flag called *wrtb0*. Although the initial state of *wrtb0* is irrelevant, it must invert upon receipt of an *AdvTFSLine* pulse.

Note:- The output addresses from Table B are registered.

27   Tag FIFO Unit (TFU)

27.1   OVERVIEW

The Tag FIFO Unit (TFU) provides the means by which data is transferred between the Tag Encoder (TE) and the HCU. By abstracting the buffering mechanism and controls from both units, the interface is clean between the data user and the data generator.

The TFU is a simple FIFO interface to the HCU. The Tag Encoder will provide support for arbitrary Y integer scaling up to 1600 dpi. X integer scaling of the tag dot data is performed at the output of the FIFO in the TFU. There is feedback to the TE from the TFU to allow stalling of the TE during a line. The TE interfaces to the TFU with a data width of 8 bits. The TFU interfaces to the HCU with a data width of 1 bit.

The depth of the TFU FIFO is chosen as 16 bytes so that the FIFO can store a single 126 dot tag.

27.1.1   Interfaces between TE, TFU and HCU

27.1.1.1   TE-TFU Interface

The interface from the TE to the TFU comprises the following signals:

- *te_tfu_wdata*, 8-bit write data.
- *te_tfu_wdatavalid*, write data valid.
- *te_tfu_wradvline*, accompanies the last valid 8-bit write data in a line.

The interface from the TFU to TE comprises the following signal:

- *tfu_te_oktowrite*, indicating to the TE that there is space available in the TFU FIFO.

The TE writes data to the TFU FIFO as long as the TFU's *tfu_te_oktowrite* output bit is set. The TE write will not occur unless data is accompanied by a data valid signal.

27.1.1.2   TFU-HCU Interface

The interface from the TFU to the HCU comprises the following signals:

- *tfu_hcu_tdata,* 1-bit data.
- *tfu_hcu_avail,* data valid signal indicating that there is data available in the TFU FIFO.

The interface from HCU to TFU comprises the following signal:

- *hcu_tfu_ready,* indicating to the TFU to supply the next dot.

27.1.1.2.1   X scaling

Tag data is replicated a scale factor (SF) number of times in the X direction to convert the final output to 1600 dpi. Unlike both the CFU and SFU, which support non-integer scaling, the scaling is integer only. Replication in the X direction is performed at the output of the TFU FIFO on a dot-by-dot basis.

To account for the case where there may be two SoPEC devices, each generating its own portion of a dot-line, the first dot in a line may not be replicated the total scale-factor number of times by an individual TFU. The dot will ultimately be scaled-up correctly with both devices doing part of the scaling, one on its lead-out and the other on its lead in.

5   Note two SoPEC TEs may be involved in producing the same byte of output tag data straddling the printhead boundary. The HCU of the left SoPEC will accept from its TE the correct amount of dots, ignoring any dots in the last byte that do not apply to its printhead. The TE of the right SoPEC will be programmed the correct number of dots into the tag and its output will be byte aligned with the left edge of the printhead.

10 27.2   DEFINITIONS OF I/O

Table 190. TFU Port List

Port Name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	SoPEC Functional clock.
Prst_n	1	In	Global reset signal.
PCU Interface data and control signals			
Pcu_adr[4:2]	2	In	PCU address bus. Only 3 bits are required to decode the address space for this block.
Pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
Tfu_pcu_datain[31:0]	32	Out	Read data bus from the TFU to the PCU.
Pcu_rwn	1	In	Common read/not-write signal from the PCU.
Pcu_tfu_sel	1	In	Block select from the PCU. When *pcu_tfu_sel* is high both *pcu_adr* and *pcu_dataout* are valid.
Tfu_pcu_rdy	1	Out	Ready signal to the PCU. When *tfu_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *tfu_pcu_datain* is valid.
TE Interface data and control signals			
Te_tfu_wdata[7:0]	8	In	Write data for TFU FIFO.

Te_tfu_wdatavalid	1	In	Write data valid signal.
Te_tfu_wradvline	1	In	Advance line signal strobed when the last byte in a line is placed on *te_tfu_wdata*
tfu_te_oktowrite	1	Out	Ready signal indicating TFU has space available in it's FIFO and is ready to be written to.
HCU Interface data and control signals			
Hcu_tfu_advdot	1	In	Signal indicating to the TFU that the HCU is ready to accept the next dot of data from TFU.
tfu_hcu_tdata	1	Out	Data from the TFU FIFO.
tfu_hcu_avail	1	Out	Signal indicating valid data available from TFU FIFO.

27.3   CONFIGURATION REGISTERS

Table 191. TFU Configuration Registers

Address TFU_Base +	register name	#bits	value on reset	description
Control registers				
0x00	Reset	1	1	A write to this register causes a reset of the TFU. This register can be read to indicate the reset state: 0 - reset in progress 1 - reset not in progress.
0x04	Go	1	see text	Writing 1 to this register starts the TFU. Writing 0 to this register halts the TFU. When *Go* is deasserted the state-machines go to their idle states but all counters and configuration registers keep their values. When *Go* is asserted all counters are reset, but configuration registers keep their values (i.e. they don't get reset). The TFU must be started before the TE is started.

				This register can be read to determine if the TFU is running (1 = running, 0 = stopped).
Setup registers (constant during processing of page)				
0x08	XScale	8	1	Tag scale factor in X direction.
0x0C	XFracScale	8	1	Tag scale factor in X direction for the first dot in a line (must be programmed to be less than or equal to *XScale*)
0x10	TEByteCount	12	0	The number of bytes to be accepted from the TE per line. Once this number of bytes have been received subsequent bytes are ignored until there is a strobe on the *te_tfu_wradvline*
0x14	HCUDotCount	16	0	The number of (optionally) x-scaled dots per line to be supplied to the HCU. Once this number has been reached the remainder of the current FIFO byte is ignored.

27.4   DETAILED DESCRIPTION

The FIFO is a simple 16-byte store with read and write pointers, and a contents store, Figure 229. 16 bytes is sufficient to store a single 126 dot tag.

Each line a total of *TEByteCount* bytes is read into the FIFO. All subsequent bytes are ignored until there is a strobe on the *te_tfu_wradvline* signal, whereupon bytes for the next line are stored. On the HCU side, a total of *HCUDotCount* dots are produced at the output. Once this count is reached any more dots in the FIFO byte currently being processed are ignored. For the first dot in the next line the start of line scale factor, *XFracScale*, is used.

The behaviour of these signals and the control signals between the TFU and the TE and HCU is detailed below.

```
// Concurrently Executed Code:
// TE always allowed to write when there's either (a)
room        or        (b)        no        room        and        all
// bytes for that line have been received.
if ((FifoCntnts != FifoMax) OR (FifoCntnts == FifoMax
and ByteToRx == 0)) then
     tfu_te_oktowrite = 1
else
     tfu_te_oktowrite = 0


// Data presented to HCU when there is (a) data in
FIFO        and        (b)        the        HCU        has        not
// received all dots for a line
if (FifoCntnts != 0) AND (BitToTx != 0) then
```

451

```
                         tfu_hcu_avail = 1
                    else
                         tfu_hcu_avail = 0


                    // Output mux of FIFO data
                    tfu_hcu_tdata = Fifo[FifoRdPnt][RdBit]


                    // Sequentially Executed Code:
                    if (te_tfu_wdatavalid == 1) AND (FifoCntnts !=
               FifoMax) AND (ByteToRx != 0) then
                         Fifo[FifoWrPnt] = te_tfu_wdata
                         FifoWrPnt ++
                         FifoContents ++
                         ByteToRx --


                    if (te_tfu_wradvline == 1) then
                         ByteToRx = TEByteCount


                    if (hcu_tfu_advdot == 1 and FifoCntnts != 0) then {
                         BitToTx ++
                         if (RepFrac == 1) then
                              RepFrac = Xscale
                              if (RdBit = 7) then
                                   RdBit = 0
                                   FifoRdPnt ++
                                   FifoContents --
                              else
                                   RdBit++
                         else
                              RepFrac--
                         if(BitToTx == 1) then {
                              RepFrac = XFracScale
                              RdBit = 0
                              FifoRdPnt ++
                              FifoContents--
                              BitToTx = HCUDotCount
                              }
                    }
```

What is not detailed above is the fact that, since this is a circular buffer, both the fifo read and
write-pointers wrap-around to zero after they reach two. Also not detailed is the fact that if there is
a change of both the read and write-pointer in the same cycle, the fifo contents counter remains
unchanged.

28   alftoner Compositor Unit (HCU)

28.1   OVERVIEW

452

The Halftoner Compositor Unit (HCU) produces dots for each nozzle in the destination printhead taking account of the page dimensions (including margins). The spot data and tag data are received in bi-level form while the pixel contone data received from the CFU must be dithered to a bi-level representation. The resultant 6 bi-level planes for each dot position on the page are then remapped to 6 output planes and output dot at a time (6 bits) to the next stage in the printing pipeline, namely the dead nozzle compensator (DNC).

28.2   DATA FLOW

Figure 230 shows a simple dot data flow high level block diagram of the HCU. The HCU reads contone data from the CFU, bi-level spot data from the SFU, and bi-level tag data from the TFU. Dither matrices are read from the DRAM via the DIU. The calculated output dot (6 bits) is read by the DNC.

The HCU is given the page dimensions (including margins), and is only started once for the page. It does not need to be programmed in between bands or restarted for each band. The HCU will stall appropriately if its input buffers are starved. At the end of the page the HCU will continue to produce 0 for all dots as long as data is requested by the units further down the pipeline (this allows later units to conveniently flush pipelined data).

The HCU performs a linear processing of dots calculating the 6-bit output of a dot in each cycle. The mapping of 6 calculated bits to 6 output bits for each dot allows for such example mappings as compositing of the spot0 layer over the appropriate contone layer (typically black), the merging of CMY into K (if K is present in the printhead), the splitting of K into CMY dots if there is no K in the printhead, and the generation of a fixative output bitstream.

28.3   DRAM STORAGE REQUIREMENTS

SoPEC allows for a number of different dither matrix configurations up to 256 bytes wide. The dither matrix is stored in DRAM. Using either a single or double-buffer scheme a line of the dither matrix must be read in by the HCU over a SoPEC line time. SoPEC must produce 13824 dots per line for A4/Letter printing which takes 13824 cycles.

The following give the storage and bandwidths requirements for some of the possible configurations of the dither matrix.

- 4 Kbyte DRAM storage required for one 64x64 (preferred) byte dither matrix
- 6.25 Kbyte DRAM storage required for one 80x80 byte dither matrix
- 16 Kbyte DRAM storage required for four 64x64 byte dither matrices
- 64 Kbyte DRAM storage required for one 256x256 byte dither matrix

It takes 4 or 8 read accesses to load a line of dither matrix into the dither matrix buffer, depending on whether we're using a single or double buffer (configured by *DoubleLineBuff* register).

28.4   IMPLEMENTATION

A block diagram of the HCU is given in Figure 231.

28.4.1   Definition of I/O

Table 192. HCU port list and description

Port name	Pins	I/O	Description

Clocks and reset			
Pclk	1	In	System clock.
prst_n	1	In	System reset, synchronous active low.
PCU interface			
pcu_hcu_sel	1	In	Block select from the PCU. When *pcu_hcu_sel* is high both *pcu_adr* and *pcu_dataout* are valid.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_adr[7:2]	6	In	PCU address bus. Only 6 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
hcu_pcu_rdy	1	Out	Ready signal to the PCU. When *hcu_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *hcu_pcu_datain* is valid.
hcu_pcu_datain[31:0]	32	Out	Read data bus to the PCU.
DIU interface			
hcu_diu_rreq	1	Out	HCU read request, active high. A read request must be accompanied by a valid read address.
diu_hcu_rack	1	In	Acknowledge from DIU, active high. Indicates that a read request has been accepted and the new read address can be placed on the address bus, *hcu_diu_radr*.
hcu_diu_radr[21:5]	17	Out	HCU read address. 17 bits wide (256-bit aligned word).
diu_hcu_rvalid	1	In	Read data valid, active high. Indicates that valid read data is now on the read data bus, *diu_data*.
diu_data[63:0]	64	In	Read data from DIU.
CFU interface			
cfu_hcu_avail	1	In	Indicates valid data present on cfu_hcu_c[3-0]data lines.
cfu_hcu_c0data[7:0]	8	In	Pixel of data in contone plane 0.
cfu_hcu_c1data[7:0]	8	In	Pixel of data in contone plane 1.
cfu_hcu_c2data[7:0]	8	In	Pixel of data in contone plane 2.
cfu_hcu_c3data[7:0]	8	In	Pixel of data in contone plane 3.
hcu_cfu_advdot	1	Out	Informs the CFU that the HCU has captured the pixel data on *cfu_hcu_c[3-0]data* lines and the CFU can now place the next pixel on the data lines.
SFU interface			
sfu_hcu_avail	1	In	Indicates valid data present on *sfu_hcu_sdata*.
sfu_hcu_sdata	1	In	Bi-level dot data.
hcu_sfu_advdot	1	Out	Informs the SFU that the HCU has captured the dot data

			on *sfu_hcu_sdata* and the SFU can now place the next dot on the data line.
TFU interface			
tfu_hcu_avail	1	In	Indicates valid data present on *tfu_hcu_tdata*.
tfu_hcu_tdata	1	In	Tag dot data.
hcu_tfu_advdot	1	Out	Informs the TFU that the HCU has captured the dot data on *tfu_hcu_tdata* and the TFU can now place the next dot on the data line.
DNC interface			
dnc_hcu_ready	1	In	Indicates that DNC is ready to accept data from the HCU.
hcu_dnc_avail	1	Out	Indicates valid data present on *hcu_dnc_data*.
hcu_dnc_data[5:0]	6	Out	Output bi-level dot data in 6 ink planes.

28.4.2   Configuration Registers

The configuration registers in the HCU are programmed via the PCU interface. Refer to section 21.8.2 on page 321 for the description of the protocol and timing diagrams for reading and writing registers in the HCU. Note that since addresses in SoPEC are byte aligned and the PCU only supports 32-bit register reads and writes, the lower 2 bits of the PCU address bus are not required to decode the address space for the HCU. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *hcu_pcu_datain*. The configuration registers of the HCU are listed in Table 193.

Table 193. HCU Registers

Address (HCU_base +)	Register Name	#bits	Value on Reset	Description
Control registers				
0x00	Reset	1	0x1	A write to this register causes a reset of the HCU.
0x04	Go	1	0x0	Writing 1 to this register starts the HCU. Writing 0 to this register halts the HCU. When *Go* is asserted all counters, flags etc. are cleared or given their initial value, but configuration registers keep their values.

				When *Go* is deasserted the state-machines go to their idle states but all counters and configuration registers keep their values. The HCU should be started *after* the CFU, SFU, TFU, and DNC. This register can be read to determine if the HCU is running (1 = running, 0 = stopped).
Setup registers (constant for during processing)				
0x10	AvailMask	4	0x0	Mask used to determine which of the dotgen units etc. are to be checked before a dot is generated by the HCU within the specified margins for the specified color plane. If the specified dotgen unit is stalled, then the HCU will also stall. See Table for bit allocation and definition.
0x14	TMMask	4	0x0	Same as *AvailMask*, but used in the top margin area before the appropriate target page is reached.
0x18	PageMarginY	32	0x0000 _ 0000	The first line considered to be off the page.
0x1C	MaxDot	16	0x0000	This is the maximum dot number - 1 present across a page. For example if a page

				contains 13824 dots, then *MaxDot* will be 13823.
0x20	TopMargin	32	0x0000_0000	The first line on a page to be considered within the target page for contone and spot data. (0 = first printed line of page)
0x24	BottomMargin	32	0x0000_0000	The first line in the target bottom margin for contone and spot data (i.e. first line after target page).
0x28	LeftMargin	16	0x0000	The first dot on a line within the target page for contone and spot data.
0x2C	RightMargin	16	0xFFFF	The first dot on a line within the target right margin for contone and spot data.
0x30	TagTopMargin	32	0x0000_0000	The first line on a page to be considered within the target page for tag data. (0 = first printed line of page)
0x34	TagBottomMargin	32	0x0000_0000	The first line in the target bottom margin for tag data (i.e. first line after target page).
0x38	TagLeftMargin	16	0x0000	The first dot on a line within the target page for tag data.
0x3C	TagRightMargin	16	0xFFFF	The first dot on a line within the target right margin for tag data.
0x44	StartDMAdr[21:5]	17	0x0_0000	Points to the first 256-bit word of the first line of the dither matrix in DRAM.
0x48	EndDMAdr[21:5]	17	0x0_0000	Points to the last address of the group of four 256-

				bit reads (or 8 if single buffering) that reads in the last line of the dither matrix.
0x4C	LineIncrement	5	0x2	The number of 256-bit words in DRAM from the start of one line of the dither matrix and the start of the next line, i.e. the value by which the DRAM address is incremented at the start of a line so that it points to the start of the next line of the dither matrix.
0x50	DMInitIndexC0	8	0x00	If using the single-buffer scheme this register represents the initial index within 256-byte dither matrix line buffer for contone plane 0. If using double-buffer scheme, only the 7 lsbs are used.
0x54	DMLwrIndexC0	8	0x00	If using the single-buffer scheme this register represents the lower index within 256-byte dither matrix line buffer for contone plane 0. If using double-buffer scheme, only the 7 lsbs are used.
0x58	DMUprIndexC0	8	0x3F	If using the single-buffer scheme this register represents the upper index within 256-byte dither matrix line buffer for contone plane 0. After reading the data at this location the index wraps to *DMLwrIndexC0*. If

				using double-buffer scheme, only the 7 lsbs are used.
0x5C	DMInitIndexC1	8	0x00	If using the single-buffer scheme this register represents the initial index within 256-byte dither matrix line buffer for contone plane 1. If using double-buffer scheme, only the 7 lsbs are used.
0x60	DMLwrIndexC1	8	0x00	If using the single-buffer scheme this register represents the lower index within 256-byte dither matrix line buffer for contone plane 1. If using double-buffer scheme, only the 7 lsbs are used.
0x64	DMUprIndexC1	8	0x3F	If using the single-buffer scheme this register represents the upper index within 256-byte dither matrix line buffer for contone plane 1. After reading the data at this location the index wraps to *DMLwrIndexC1*. If using double-buffer scheme, only the 7 lsbs are used.
0x68	DMInitIndexC2	8	0x00	If using the single-buffer scheme this register represents the initial index within 256-byte dither matrix line buffer for contone plane 2. If using double-buffer scheme, only the 7 lsbs are used.

0x6C	DMLwrIndexC2	8	0x00	If using the single-buffer scheme this register represents the lower index within 256-byte dither matrix line buffer for contone plane 2. If using double-buffer scheme, only the 7 lsbs are used.
0x70	DMUprIndexC2	8	0x3F	If using the single-buffer scheme this register represents the upper index within 256-byte dither matrix line buffer for contone plane 2. After reading the data at this location the index wraps to *DMLwrIndexC2*. If using double-buffer scheme, only the 7 lsbs are used.
0x74	DMInitIndexC3	8	0x00	If using the single-buffer scheme this register represents the initial index within 256-byte dither matrix line buffer for contone plane 3. If using double-buffer scheme, only the 7 lsbs are used.
0x78	DMLwrIndexC3	8	0x00	If using the single-buffer scheme this register represents the lower index within 256-byte dither matrix line buffer for contone plane 3. If using double-buffer scheme, only the 7 lsbs are used.
0x7C	DMUprIndexC3	8	0x3F	If using the single-buffer scheme this register represents the upper index within 256-byte

				dither matrix line buffer for contone plane 3. After reading the data at this location the index wraps to *DMLwrIndexC3*. If using double-buffer scheme, only the 7 lsbs are used.
0x80	DoubleLineBuf	1	0x1	Selects the dither line buffer mode to be single or double buffer. 0 - single line buffer mode 1 - double line buffer mode
0x84 to 0x98	IOMappingLo	6x32	0x0000 _ 0000	The dot reorg mapping for output inks 0 to 5. For each ink's 64-bit IOMapping value, *IOMappingLo* represents the low order 32 bits.
0x9C to 0xB0	IOMappingHi	6x32	0x0000 _ 0000	The dot reorg mapping for output inks 0 to 5. For each ink's 64-bit IOMapping value, *IOMappingHi* represents the high order 32 bits.
0xB4 to 0xC0	cpConstant	4x8	0x00	The constant contone value to output for contone plane N when printing in the margin areas of the page. This value will typically be 0.
0xC4	sConstant	1	0x0	The constant bi-level value to output for spot when printing in the margin areas of the page. This value will typically be 0.
0xC8	tConstant	1	0x0	The constant bi-level

				value to output for tag data when printing in the margin areas of the page. This value will typically be 0.
0xCC	DitherConstant	8	0xFF	The constant value to use for dither matrix when the dither matrix is not available, i.e. when the signal *dm_avail* is 0. This value will typically be 0xFF so that *cpConstant* can easily be 0x00 or 0xFF without requiring a dither matrix (*DitherConstant* is primarily used for threshold dithering in the margin areas).
Debug registers (read only)				
0xD0	HcuPortsDebug	14	N/A	Bit 13 = *tfu_hcu_avail* Bit 12 = *hcu_tfu_advdot* Bit 11 = *sfu_hcu_avail* Bit 10 = *hcu_sfu_advdot* Bit 9 = *cfu_hcu_avail* Bit 8 = *hcu_cfu_advdot* Bit 7 = *dnc_hcu_ready* Bit 6 = *hcu_dnc_avail* Bits 5-0 = *hcu_dnc_data*
0xD4	HcuDotgenDebug	15	N/A	Bit 14 = *after_top_margin* Bit 13 = *in_tag_target_page* Bit 12 = *in_target_page* Bit 11 = *tp_avail* Bit 10 = *s_avail* Bit 9 = *cp_avail* Bit 8 = *dm_avail* Bit 7 = *advdot* Bits 5-0 =

				[*tp,s,cp3,cp2,cp1,cp0*] (i.e. 6 bit input to dot reorg units)
0xD8	HcuDitherDebug1	17	N/A	Bit 17 = *advdot* Bit 16 = *dm_avail* Bit 15-8 = *cp1_dither_val* Bits 7-0 = *cp0_dither_val*
0xDC	HcuDitherDebug2	17	N/A	Bit 17 = *advdot* Bit 16 = *dm_avail* Bit 15-8 = *cp3_dither_val* Bits 7-0 = *cp2_dither_vall*

28.4.3 Control unit

The control unit is responsible for controlling the overall flow of the HCU. It is responsible for determining whether or not a dot will be generated in a given cycle, and what dot will actually be generated - including whether or not the dot is in a margin area, and what dither cell values should be used at the specific dot location. A block diagram of the control unit is shown in Figure 232.

The inputs to the control unit are a number of avail flags specifying whether or not a given dotgen unit is capable of supplying 'real' data in this cycle. The term 'real' refers to data generated from external sources, such as contone line buffers, bi-level line buffers, and tag plane buffers. Each dotgen unit informs the control unit whether or not a dot can be generated this cycle from real data. It must also check that the DNC is ready to receive data.

The contone/spot margin unit is responsible for determining whether the current dot coordinate is within the target contone/spot margins, and the tag margin unit is responsible for determining whether the current dot coordinate is within the target tag margins.

The dither matrix table interface provides the interface to DRAM for the generation of dither cell values that are used in the halftoning process in the contone dotgen unit.

28.4.3.1 Determine advdot

The HCU does not always require contone planes, bi-level or tag planes in order to produce a page. For example, a given page may not have a bi-level layer, or a tag layer. In addition, the contone and bi-level parts of a page are only required within the contone and bi-level page margins, and the tag part of a page is only required within the tag page margins. Thus output dots can be generated without contone, bi-level or tag data before the respective top margins of a page has been reached, and 0s are generated for all color planes after the end of the page has been reached (to allow later stages of the printing pipeline to flush).

Consequently the HCU has an *AvailMask* register that determines which of the various input avail flags should be taken notice of during the production of a page from the first line of the target page, and a *TMMask* register that has the same behaviour, but is used in the lines before the target page has been reached (i.e. inside the target top margin area). The dither matrix mask bit *TMask[0]* is the exception, it applies to all margins areas not just the top margin. Each bit in the

AvailMask refers to a particular avail bit: if the bit in the *AvailMask* register is set, then the corresponding *avail* bit must be 1 for the HCU to advance a dot. The bit to avail correspondence is shown in Table 194. Care should be taken with *TMMask* - if the particular data is not available after the top margin has been reached, then the HCU will stall. Note that the *avail* bits for contone and spot colors are ANDed with *in_target_page* after the target page area has been reached to allow dot production in the contone/spot margin areas without needing any data in the CFU and SFU. The *avail* bit for tag color is ANDed with *in_tag_target_page* after the target tag page area has been reached to allow dot production in the tag margin areas without needing any data in the TFU.

Table 194. Correspondence between bit in AvailMask and avail flag

bit # in AvailMask	avail flag	description
0	dm_avail	dither matrix data available
1	cp_avail	contone pixels available
2	s_avail	spot color available
3	tp_avail	tag plane available

Each of the input *avail* bits is processed with its appropriate mask bit and the *after_top_margin* flag (note the dither matrix is the exception it is processed with *in_target_page*). The output bits are ANDed together along with *Go* and *output_buff_full* (which specifies whether the output buffer is ready to receive a dot in this cycle) to form the output bit *advdot*. We also generate *wr_advdot*. In this way, if the output buffer is full or any of the specified avail flags is clear, the HCU will stall. When the end of the page is reached, *in_page* will be deasserted and the HCU will continue to produce 0 for all dots as long as the DNC requests data. A block diagram of the determine advdot unit is shown in Figure 233.

The advance dot block also determines if current page needs dither matrix, it indicates to the dither matrix table interface block via the *dm_read_enable* signal. If no dither is required in the margins or in the target page then *dm_read_enable* will be 0 and no dither will be read in for this page.

28.4.3.2 Position unit

The position unit is responsible for outputting the position of the current dot (*curr_pos*, *curr_line*) and whether or not this dot is the last dot of a line (*advline*). Both *curr_pos* and *curr_line* are set to 0 at reset or when *Go* transitions from 0 to 1. The position unit relies on the *advdot* input signal to advance through the dots on a page. Whenever an *advdot* pulse is received, *curr_pos* gets incremented. If *curr_pos* equals *max_dot* then an *advline* pulse is generated as this is the last dot in a line, *curr_line* gets incremented, and the *curr_pos* is reset to 0 to start counting the dots for the next line.

The position unit also generates a filtered version of *advline* called *dm_advline* to indicate to the dither matrix pointers to increment to the next line. The *dm_advline* is only incremented when dither is required for that line.

```
if ((after_top_margin AND avail_mask[0]) OR tm_mask[0]) then
    dm_advline = advline
else
    dm_advline = 0
```

28.4.3.3 Margin unit

The responsibility of the margin unit is to determine whether the specific dot coordinate is within the page at all, within the target page or in a margin area (see Figure 234). This unit is instantiated for both the contone/spot margin unit and the tag margin unit.

The margin unit takes the current dot and line position, and returns three flags.

- the first, *in_page* is 1 if the current dot is within the page, and 0 if it is outside the page.

- the second flag, *in_target_page*, is 1 if the dot coordinate is within the target page area of the page, and 0 if it is within the target top/left/bottom/right margins.

- the third flag, after_top_margin, is 1 if the current dot is below the target top margin, and 0 if it is within the target top margin.

A block diagram of the margin unit is shown in Figure 235.

28.4.3.4 Dither matrix table interface

The dither matrix table interface provides the interface to DRAM for the generation of dither cell values that are used in the halftoning process in the contone dotgen unit. The control flag *dm_read_enable* enables the reading of the dither matrix table line structure from DRAM. If *dm_read_enable* is 0, the dither matrix is not specified in DRAM and no DRAM accesses are attempted. The dither matrix table interface has an output flag *dm_avail* which specifies if the current line of the specified matrix is available. The HCU can be directed to stall when *dm_avail* is 0 by setting the appropriate bit in the HCU's *AvailMask* or *TMMask* registers. When *dm_avail* is 0 the value in the *DitherConstant* register is used as the dither cell values that are output to the contone dotgen unit.

The dither matrix table interface consists of a state machine that interfaces to the DRAM interface, a dither matrix buffer that provides dither matrix values, and a unit to generate the addresses for reading the buffer. Figure 236 shows a block diagram of the dither matrix table interface.

28.4.3.5 Dither data structure in DRAM

The dither matrix is stored in DRAM in 256-bit words, transferred to the HCU in 64-bit words and consumed by the HCU in bytes. Table 195 shows the 64-bit words mapping to 256-bit word addresses, and Table 196 shows the 8-bits dither value mapping in the 64-bits word.

Table 195. Dither Data stored in DRAM

Address[21:5]	Data[255:0]			
00000	D3 [255:192]	D2 [191:128]	D1 [127:64]	D0 [63:0]
00001	D7 [255:192]	D6 [191:128]	D5 [127:64]	D4 [63:0]
00010	D11	D10	D9	D8

	[255:192]	[191:128]	[127:64]	[63:0]
00011	D15 [255:192]	D14 [191:128]	D13 [127:64]	D12 [63:0]
00100	D19 [255:192]	D18 [191:128]	D17 [127:64]	D16 [63:0]
etc				

When the HCU first requests data from DRAM, the 64-bits word transfer order will be D0,D1,D2,D3. On the second request the transfer order will be D4,D5,D6,D7 and so on for other requests.

Table 196. Dither data stored in HCUs line buffer

Dither index[7:0]	Data[7:0]	Dither index[7:0]	Data[7:0]	Dither index[7:0]	Data[7:0]
00	D0[7:0]	10	D2[7:0]	20	D4[7:0]
01	D0[15:8]	11	D2[15:8]	21	D4[15:8]
02	D0[23:16]	12	D2[23:16]	22	D4[23:16]
03	D0[31:24]	13	D2[32:24]	23	D4[31:24]
04	D0[39:32]	14	D2[39:32]	24	D4[39:32]
05	D0[47:40]	15	D2[47:40]	25	D4[47:40]
06	D0[55:48]	16	D2[55:48]	26	D4[55:48]
07	D0[63:56]	17	D2[63:56]	27	D4[63:56]
08	D1[7:0]	18	D3[7:0]	28	D5[7:0]
09	D1[15:8]	19	D3[15:8]	29	D5[15:8]
0A	D1[23:16]	1A	D3[23:16]	2A	D5[23:16]
0B	D1[31:24]	1B	D3[31:24]	2B	D5[31:24]
0C	D1[39:32]	1C	D3[39:32]	2C	D5[39:32]
0D	D1[47:40]	1D	D3[47:40]	2D	D5[47:40]
0E	D1[55:48]	1E	D3[55:48]	2E	D5[55:48]
0F	D1[63:56]	1F	D3[63:56]	2F	D5[63:56]
				etc.	etc.

28.4.3.5.1   Dither matrix buffer

The state machine loads dither matrix table data a line at a time from DRAM and stores it in a buffer. A single line of the dither matrix is either 256 or 128 8-bit entries, depending on the programmable bit *DoubleLineBuf*. If this bit is enabled, a double-buffer mechanism is employed such that while one buffer is read from for the current line's dither matrix data (8 bits representing a single dither matrix entry), the other buffer is being written to with the next line's dither matrix data (64-bits at a time). Alternatively, the single buffer scheme can be used, where the data must be loaded at the end of the line, thus incurring a delay.

466

The single/double buffer is implemented using a 256 byte 3-port register array, two reads, one write port, with the reads clocked at double the system clock rate (320MHz) allowing 4 reads per clock cycle.

The dither matrix buffer unit also provides the mechanism for keeping track of the current read and write buffers, and providing the mechanism such that a buffer cannot be read from until it has been written to. In this case, each buffer is a line of the dither matrix, i.e. 256 or 128 bytes.

The dither matrix buffer maintains a read and write pointer for the dither matrix. The output value *dm_avail* is derived by comparing the read and write pointers to determine when the dither matrix is not empty. The write pointer *wr_adr* is incremented each time a 64-bit word is written to the dither matrix buffer and the read pointer *rd_ptr* is incremented each time *dm_advline* is received. If *double_line_buf* is 0 the *rd_ptr* will increment by 2, otherwise it will increment by 1. If the dither matrix buffer is full then no further writes will be allowed (*buff_full* =1), or if the buffer is empty no further buffer reads are allowed (*buff_emp*=1).

The read addresses are byte aligned and are generated by the read address generator. A single dither matrix entry is represented by 8 bits and an entry is read for each of the four contone planes in parallel. If double buffer is used (*double_line_buf=1*) the read address is derived from 7-bit address from the read address generator and 1-bit from the read pointer. If *double_line_buf=0* then the read address is the full 8-bits from the read address generator.

```
if (double_line_buf == 1 )then
   read_port[7:0]    =    {rd_ptr[0],rd_adr[6:0]}          //
concatenation
else
   read_port[7:0] = rd_adr[7:0]
```

28.4.3.5.2   Read address generator

For each contone plane there is a initial, lower and upper index to be used when reading dither cell values from the dither matrix double buffer. The read address for each plane is used to select a byte from the current 256-byte read buffer. When *Go* gets set (0 to 1 transition), or at the end of a line, the read addresses are set to their corresponding initial index. Otherwise, the read address generator relies on *advdot* to advance the addresses within the inclusive range specified the lower and upper indices, represented by the following pseudocode:

```
if (advdot == 1) then
   if (advline == 1) then
      rd_adr = dm_init_index
   elsif (rd_adr == dm_upr_index) then
      rd_adr = dm_lwr_index
   else
      rd_adr ++
else
   rd_adr = rd_adr
```

28.4.3.5.3   State machine

The dither matrix is read from DRAM in single 256-bit accesses, receiving the data from the DIU over 4 clock cycles (64-bits per cycle). The protocol and timing for read accesses to DRAM is described in section 20.9.1 on page 240. Read accesses to DRAM are implemented by means of the state machine described in Figure 238.

5   All counters and flags should be cleared after reset or when *Go* transitions from 0 to 1. While the *Go* bit is 1, the state machine relies on the *dm_read_enable* bit to tell it whether to attempt to read dither matrix data from DRAM. When *dm_read_enable* is clear, the state machine does nothing and remains in the idle state. When *dm_read_enable* is set, the state machine continues to load dither matrix data, 256-bits at a time (received over 4 clock cycles, 64 bits per cycle), while there

10   is space available in the dither matrix buffer, (*buff_full* !=1).

The read address and *line_start_adr* are initially set to *start_dm_adr*. The read address gets incremented after each read access. It takes 4 or 8 read accesses to load a line of dither matrix into the dither matrix buffer, depending on whether we're using a single or double buffer. A count is kept of the accesses to DRAM. When a read access completes and *access_count* equals 3 or

15   7, a line of dither matrix has just been loaded from and the read address is updated to *line_start_adr* plus *line_increment* so it points to the start of the next line of dither matrix. (*line_start_adr* is also updated to this value). If the read address equals *end_dm_adr* then the next read address will be *start_dm_adr*, thus the read address wraps to point to the start of the area in DRAM where the dither matrix is stored.

20   The write address for the dither matrix buffer is implemented by means of a modulo-32 counter that is initially set to 0 and incremented when *diu_hcu_rvalid* is asserted.

Figure 237 shows an example of setting *start_dm_adr* and *end_dm_adr* values in relation to the line increment and double line buffer settings. The calculation of *end_dm_adr* is

```
                    // end_dm_adr calculation
25                  dm_height = Dither matrix height in lines
                    if (double_line_buf == 1)        //
                      end_dm_adr[21:5]  =  start_dm_adr[21:5]  +  (((dm_height  -
                    1)*line_inc) + 3) << 5)
                    else
30                    end_dm_adr[21:5]  =  start_dm_adr[21:5]  +  (((dm_height  -
                    1)*line_inc) + 7) << 5)
```

28.4.4   Contone dotgen unit

The contone dotgen unit is responsible for producing a dot in up to 4 color planes per cycle. The contone dotgen unit also produces a *cp_avail* flag which specifies whether or not contone pixels

35   are currently available, and the output *hcu_cfu_advdot* to request the CFU to provide the next contone pixel in up to 4 color planes.

The block diagram for the contone dotgen unit is shown in Figure 239.

A dither unit provides the functionality for dithering a single contone plane. The contone image is only defined within the contone/spot margin area. As a result, if the input flag *in_target_page* is 0,

40   then a constant contone pixel value is used for the pixel instead of the contone plane.

The resultant contone pixel is then halftoned. The dither value to be used in the halftoning process is provided by the control data unit. The halftoning process involves a comparison between a pixel value and its corresponding dither value. If the 8-bit contone value is *greater than or equal* to the 8-bit dither matrix value a 1 is output. If not, then a 0 is output. This means each entry in the dither matrix is in the range 1-255 (0 is not used).

Note that constant use is dependant on the *in_target_page* signal only, if *in_target_page* is 1 then the *cfu_hcu_c*_data* should be allowed to pass through, regardless of the stalling behaviour or the *avail_mask[1]* setting. This allows a constant value to be setup on the CFU output data, and the use of different constants while inside and outside the target page. The *hcu_cfu_advdot* will always be zero if the *avail_mask[1]* is zero.

28.4.5   Spot dotgen unit

The spot dotgen unit is responsible for producing a dot of bi-level data per cycle. It deals with bi-level data (and therefore does not need to halftone) that comes from the LBD via the SFU. Like the contone layer, the bi-level spot layer is only defined within the contone/spot margin area. As a result, if input flag *in_target_page* is 0, then a constant dot value (typically this would be 0) is used for the output dot.

The spot dotgen unit also produces a *s_avail* flag which specifies whether or not spot dots are currently available for this spot plane, and the output *hcu_sfu_advdot* to request the SFU to provide the next bi-level data value. The spot dotgen unit can be represented by the following pseudocode:

```
s_avail = sfu_hcu_avail

if (in_target_page == 1 AND avail_mask[2] == 0 )OR
    (in_target_page == 0) then
  hcu_sfu_advdot = 0
else
  hcu_sfu_advdot = advdot

if (in_target_page == 1) then
  sp = sfu_hcu_sdata
else
  sp = sp_constant
```

Note that constant use is dependant on the *in_target_page* signal only, if *in_target_page* is 1 then the *sfu_hcu_data* should be allowed to pass through, regardless of the stalling behaviour or the *avail_mask* setting. This allows a constant value to be setup on the SFU output data, and the use of different constants while inside and outside the target page. The *hcu_sfu_advdot* will always be zero if the *avail_mask[2]* is zero.

28.4.6   Tag dotgen unit

This unit is very similar to the spot dotgen unit (see Section 28.4.5) in that it deals with bi-level data, in this case from the TE via the TFU. The tag layer is only defined within the tag margin area. As a result, if input flag *in_tag_target_page* is 0, then a constant dot value, *tp_constant* (typically this would be 0), is used for the output dot. The tagplane dotgen unit also produces a

tp_avail flag which specifies whether or not tag dots are currently available for the tagplane, and the output *hcu_tfu_advdot* to request the TFU to provide the next bi-level data value.

The *hcu_tfu_advdot* generation is similar to the SFU and CFU, except it depends only on *in_target_page* and *advdot*. It does not take into account the avail mask when inside the target page.

28.4.7   Dot reorg unit

The dot reorg unit provides a means of mapping the bi-level dithered data, the spot0 color, and the tag data to output inks in the actual printhead. Each dot reorg unit takes a set of 6 1-bit inputs and produces a single bit output that represents the output dot for that color plane.

The output bit is a logical combination of any or all of the input bits. This allows the spot color to be placed in any output color plane (including infrared for testing purposes), black to be merged into cyan, magenta and yellow (in the case of no black ink in the Memjet printhead), and tag dot data to be placed in a visible plane. An output for fixative can readily be generated by simply combining desired input bits.

The dot reorg unit contains a 64-bit lookup to allow complete freedom with regards to mapping. Since all possible combinations of input bits are accounted for in the 64 bit lookup, a given dot reorg unit can take the mapping of other reorg units into account. For example, a black plane reorg unit may produce a 1 only if the contone plane 3 or spot color inputs are set (this effectively composites black bi-level over the contone). A fixative reorg unit may generate a 1 if any 2 of the output color planes is set (taking into account the mappings produced by the other reorg units).

If dead nozzle replacement is to be used (see section 29.4.2 on page 473), the dot reorg can be programmed to direct the dots of the specified color into the main plane, and 0 into the other. If a nozzle is then marked as dead in the DNC, swapping the bits between the planes will result in 0 in the dead nozzle, and the required data in the other plane.

If dead nozzle replacement is to be used, and there are no tags, the TE can be programmed with the position of dead nozzles and the resultant pattern used to direct dots into the specified nozzle row. If only fixed background TFS is to be used, a limited number of nozzles can be replaced. If variable tag data is to be used to specify dead nozzles, then large numbers of dead nozzles can be readily compensated for.

The dot reorg unit can be used to average out the nozzle usage when two rows of nozzles share the same ink and tag encoding is not being used. The TE can be programmed to produce a regular pattern (e.g. 0101 on one line, and 1010 on the next) and this pattern can be used as a directive as to direct dots into the specified nozzle row.

Each reorg unit contains a 64-bit *IOMapping* value programmable as two 32-bit HCU registers, and a set of selection logic based on the 6-bit dot input (2^6 = 64 bits), as shown in Figure 240. The mapping of input bits to each of the 6 selection bits is as defined in Table 197.

Table 197. Mapping of input bits to 6 selection bits

address bit of lookup	tied to	likely interpretation

0	bi-level dot from contone layer 0	cyan
1	bi-level dot from contone layer 1	magenta
2	bi-level dot from contone layer 2	yellow
3	bi-level dot from contone layer 3	black
4	bi-level spot0 dot	black
5	bi-level tag dot	infra-red

28.4.8   Output buffer

The output buffer de-couples the stalling behaviour of the feeder units from the stalling behaviour of the DNC. The larger the buffer the greater de-coupling. Currently the output buffer size is 2, but could be increased if needed at the cost of extra area.

If the *Go* bit is set to 0 no read or write of the output buffer is permitted. On a low to high transition of the *Go* bit the contents of the output buffer are cleared.

The output buffer also implements the interface logic to the DNC. If there is data in the output buffer the *hcu_dnc_avail* signal will be 1, otherwise is will be 0. If both *hcu_dnc_avail* and *dnc_hcu_ready* are 1 then data is read from the output buffer.

On the write side if there is space available in the output buffer the logic indicates to the control unit via the *output_buff_full* signal. The control unit will then allow writes to the output buffer via the *wr_advdot* signal. If the writes to the output buffer are after the end of a page (indicated by *in_page* equal to 0) then all dots written into the output buffer are set to zero.

28.4.8.1   HCU to DNC interface

Figure 241 shows the timing diagram and representative logic of the HCU to DNC interface. The *hcu_dnc_avail* signal indicate to the DNC that the HCU has data available. The *dnc_hcu_ready* signal indicates to the HCU that the DNC is ready to accept data. When both signals are high data is transferred from the HCU to the DNC. Once the HCU indicates it has data available (setting the *hcu_dnc_avail* signal high) it can only set the *hcu_dnc_avail* low again after a dot is accepted by the DNC.

28.4.9   Feeder to HCU interfaces

Figure 242 shows the feeder unit to HCU interface timing diagram, and Figure 243 shows representative logic of the interface with the register positions. *sfu_hcu_data* and *sfu_hcu_avail* are always registered while the *sfu_hcu_advdot* is not. The *hcu_sfu_avail* signal indicates to the HCU that the feeder unit has data available, and *sfu_hcu_advdot* indicates to the feeder unit that the HCU has captured the last dot. The HCU can never produce an advance dot pulse while the avail is low. The diagrams show the example of the SFU to HCU interface, but the same interface is used for the other feeder units TFU and CFU.

29   Dead Nozzle Compensator (DNC)

29.1   OVERVIEW

The Dead Nozzle Compensator (DNC) is responsible for adjusting Memjet dot data to take account of non-functioning nozzles in the Memjet printhead. Input dot data is supplied from the

HCU, and the corrected dot data is passed out to the DWU. The high level data path is shown by the block diagram in Figure 244.

The DNC compensates for a dead nozzles by performing the following operations:

- Dead nozzle removal, i.e. turn the nozzle off
- Ink replacement by direct substitution i.e. K -> K
- Ink replacement by indirect substitution i.e. K -> CMY
- Error diffusion to adjacent nozzles
- Fixative corrections

The DNC is required to efficiently support up to 5% dead nozzles, under the expected DRAM bandwidth allocation, with no restriction on where dead nozzles are located and handle any fixative correction due to nozzle compensations. Performance must degrade gracefully after 5% dead nozzles.

29.2   DEAD NOZZLE IDENTIFICATION

Dead nozzles are identified by means of a position value and a mask value. Position information is represented by a 10-bit delta encoded format, where the 10-bit value defines the number of dots between dead nozzle columns[19]. With the delta information it also reads the 6-bit dead nozzle mask (*dn_mask*) for the defined dead nozzle position. Each bit in the *dn_mask* corresponds to an ink plane. A set bit indicates that the nozzle for the corresponding ink plane is dead. The dead nozzle table format is shown in Figure 245. The DNC reads dead nozzle information from DRAM in single 256-bit accesses. A 10-bit delta encoding scheme is chosen so that each table entry is 16 bits wide, and 16 entries fit exactly in each 256-bit read. Using 10-bit delta encoding means that the maximum distance between dead nozzle columns is 1023 dots. It is possible that dead nozzles may be spaced further than 1023 dots from each other, so a null dead nozzle identifier is required. A null dead nozzle identifier is defined as a 6-bit *dn_mask* of all zeros. These null dead nozzle identifiers should also be used so that:

- the dead nozzle table is a multiple of 16 entries (so that it is aligned to the 256-bit DRAM locations)
- the dead nozzle table spans the complete length of the line, i.e. the first entry dead nozzle table should have a delta from the first nozzle column in a line and the last entry in the dead nozzle table should correspond to the last nozzle column in a line.

Note that the DNC deals with the width of a page. This may or may not be the same as the width of the printhead (the PHI may introduce some margining to the page so that its dot output matches the width of the printhead). Care must be taken when programming the dead nozzle table so that dead nozzle positions are correctly specified with respect to the page and printhead.

29.3   DRAM STORAGE AND BANDWIDTH REQUIREMENT

The memory required is largely a factor of the number of dead nozzles present in the printhead (which in turn is a factor of the printhead size). The DNC is required to read a 16-bit entry from the

---

[19]for a 10-bit delta value of *d*, if the current column *n* is a dead nozzle column then the next dead nozzle column is given by *n* + (*d* + 1).

dead nozzle table for every dead nozzle. Table 198 shows the DRAM storage and average[20] bandwidth requirements for the DNC for different percentages of dead nozzles and different page sizes.

Table 198. Dead Nozzle storage and average bandwidth requirements

Page size	% Dead Nozzles	Dead nozzle table	
		Memory (KBytes)	Bandwidth (bits/cycle)
A4[a]	5%	1.4[c]	0.8[d]
	10%	2.7	1.6
	15%	4.1	2.4
A3[b]	5%	1.9	0.8
	10%	3.8	1.6
	15%	5.7	2.4

a. Bi-lithic printhead has 13824 nozzles per color providing full bleed printing for A4/Letter
b. Bi-lithic printhead has 19488 nozzles per color providing full bleed printing for A3
c. 16 bits x 13824 nozzles x 0.05 dead
d. (16 bits read / 20 cycles) = 0.8 bits/cycle

29.4   NOZZLE COMPENSATION

DNC receives 6 bits of dot information every cycle from the HCU, 1 bit per color plane. When the dot position corresponds to a dead nozzle column, the associated 6-bit *dn_mask* indicates which ink plane(s) contains a dead nozzle(s). The DNC first deletes dots destined for the dead nozzle. It then replaces those dead dots, either by placing the data destined for the dead nozzle into an adjacent ink plane (direct substitution) or into a number of ink planes (indirect substitution). After ink replacement, if a dead nozzle is made active again then the DNC performs error diffusion. Finally, following the dead nozzle compensation mechanisms the fixative, if present, may need to be adjusted due to new nozzles being activated, or dead nozzles being removed.

29.4.1   Dead nozzle removal

If a nozzle is defined as dead, then the first action for the DNC is to turn off (zeroing) the dot data destined for that nozzle. This is done by a bit-wise ANDing of the inverse of the *dn_mask* with the dot value.

29.4.2   Ink replacement

Ink replacement is a mechanism where data destined for the dead nozzle is placed into an adjacent ink plane of the same color (direct substitution, i.e. K -> $K_{alternative}$), or placed into a number of ink planes, the combination of which produces the desired color (indirect substitution, i.e. K -> CMY). Ink replacement is performed by filtering out ink belonging to nozzles that are

---

[20]Average bandwidth assumes an even spread of dead nozzles. Clumps of dead nozzles may cause delays due to insufficient available DRAM bandwidth. These delays will occur every line causing an accumulative delay over a page.

dead and then adding back in an appropriately calculated pattern. This two step process allows the optional re-inclusion of the ink data into the original dead nozzle position to be subsequently error diffused. In the general case, fixative data destined for a dead nozzle should not be left active intending it to be later diffused.

5   The ink replacement mechanism has 6 ink replacement patterns, one per ink plane, programmable by the CPU. The dead nozzle mask is ANDed with the dot data to see if there are any planes where the dot is active but the corresponding nozzle is dead. The resultant value forms an enable, on a per ink basis, for the ink replacement process. If replacement is enabled for a particular ink, the values from the corresponding replacement pattern register are ORed into the

10   dot data. The output of the ink replacement process is then filtered so that error diffusion is only allowed for the planes in which error diffusion is enabled. The output of the ink replacement logic is ORed with the resultant dot after dead nozzle removal. See Figure n page565 on page **Error! Bookmark not defined.** for implementation details.

For example if we consider the printhead color configuration C,M,Y,K_1,K_2,IR and the input dot

15   data from the HCU is b101100. Assuming that the K_1 ink plane and IR ink plane for this position are dead so the dead nozzle mask is b000101. The DNC first removes the dead nozzle by zeroing the K_1 plane to produce b101000. Then the dead nozzle mask is ANDed with the dot data to give b000100 which selects the ink replacement pattern for K_1 (in this case the ink replacement pattern for K_1 is configured as b000010, i.e. ink replacement into the K_2 plane). Providing error

20   diffusion for K_2 is enabled, the output from the ink replacement process is b000010. This is ORed with the output of dead nozzle removal to produce the resultant dot b101010. As can be seen the dot data in the defective K_1 nozzle was removed and replaced by a dot in the adjacent K_2 nozzle in the same dot position, i.e. direct substitution.

In the example above the K_1 ink plane could be compensated for by indirect substitution, in which

25   case ink replacement pattern for K_1 would be configured as b111000 (substitution into the CMY color planes), and this is ORed with the output of dead nozzle removal to produce the resultant dot b111000. Here the dot data in the defective K_1 ink plane was removed and placed into the CMY ink planes.

29.4.3   Error diffusion

30   Based on the programming of the lookup table the dead nozzle may be left active after ink replacement. In such cases the DNC can compensate using error diffusion. Error diffusion is a mechanism where dead nozzle dot data is diffused to adjacent dots.

When a dot is active and its destined nozzle is dead, the DNC will attempt to place the data into an adjacent dot position, if one is inactive. If both dots are inactive then the choice is arbitrary, and

35   is determined by a pseudo random bit generator. If both neighbor dots are already active then the bit cannot be compensated by diffusion.

Since the DNC needs to look at neighboring dots to determine where to place the new bit (if required), the DNC works on a set of 3 dots at a time. For any given set of 3 dots, the first dot

received from the HCU is referred to as dot A, and the second as dot B, and the third as dot C. The relationship is shown in Figure 246.

For any given set of dots ABC, only B can be compensated for by error diffusion if B is defined as dead. A 1 in dot B will be diffused into either dot A or dot C if possible. If there is already a 1 in dot

5   A or dot C then a 1 in dot B cannot be diffused into that dot.

The DNC must support adjacent dead nozzles. Thus if dot A is defined as dead and has previously been compensated for by error diffusion, then the dot data from dot B should not be diffused into dot A. Similarly, if dot C is defined as dead, then dot data from dot B should not be diffused into dot C.

10   Error diffusion should not cross line boundaries. If dot B contains a dead nozzle and is the first dot in a line then dot A represents the last dot from the previous line. In this case an active bit on a dead nozzle of dot B should not be diffused into dot A. Similarly, if dot B contains a dead nozzle and is the last dot in a line then dot C represents the first dot of the next line. In this case an active bit on a dead nozzle of dot B should not be diffused into dot C.

15   Thus, as a rule, a 1 in dot B cannot be diffused into dot A if

- a 1 is already present in dot A,
- dot A is defined as dead,
- or dot A is the last dot in a line.

Similarly, a 1 in dot B cannot be diffused into dot C if

20   - a 1 is already present in dot C,
- dot C is defined as dead,
- or dot C is the first dot in a line.

If B is defined to be dead and the dot value for B is 0, then no compensation needs to be done and dots A and C do not need to be changed.

25   If B is defined to be dead and the dot value for B is 1, then B is changed to 0 and the DNC attempts to place the 1 from B into either A or C:

- If the dot can be placed into both A and C, then the DNC must choose between them. The preference is given by the current output from the random bit generator, 0 for "prefer left" (dot A) or 1 for "prefer

30   right" (dot C).

- If dot can be placed into only one of A and C, then the 1 from B is placed into that position.
- If dot cannot be placed into either one of A or C, then the DNC cannot place the dot in either position.

35

Table 199. Error Diffusion Truth Table when dot B is dead

Input				Output		
A OR A dead	B	C OR C dead	Rand a	A	B	C

OR A last in line		OR C first in line				
0	0	0	X	A input	0	C input
0	0	1	X	A input	0	C input
0	1	0	0	1`b	0	C input
0	1	0	1	A input	0	1
0	1	1	X	1	0	C input
1	0	0	X	A input	0	C input
1	0	1	X	A input	0	C input
1	1	0	X	A input	0	1
1	1	1	X	A input	0	C input

Table 199 shows the truth table for DNC error diffusion operation when dot B is defined as dead.

a. Output from random bit generator. Determines direction of error diffusion (0 = left, 1 = right)

b. Bold emphasis is used to show the DNC inserted a 1

The random bit value used to arbitrarily select the direction of diffusion is generated by a 32-bit maximum length random bit generator. The generator generates a new bit for each dot in a line regardless of whether the dot is dead or not. The random bit generator can be initialized with a 32-bit programmable seed value.

29.4.4 Fixative correction

After the dead nozzle compensation methods have been applied to the dot data, the fixative, if present, may need to be adjusted due to new nozzles being activated, or dead nozzles being removed. For each output dot the DNC determines if fixative is required (using the *FixativeRequiredMask* register) for the new compensated dot data word and whether fixative is activated already for that dot. For the DNC to do so it needs to know the color plane that has fixative, this is specified by the *FixativeMask1* configuration register. Table 200 indicates the actions to take based on these calculations.

Table 200. Truth table for fixative correction

Fixative Present	Fixative required	Action
1	1	Output dot as is.
1	0	Clear fixative plane.
0	1	Attempt to add fixative.
0	0	Output dot as is.

The DNC also allows the specification of another fixative plane, specified by the *FixativeMask2* configuration register, with *FixativeMask1* having the higher priority over *FixativeMask2*. When attempting to add fixative the DNC first tries to add it into the planes defined by *FixativeMask1*. However, if any of these planes is dead then it tries to add fixative by placing it into the planes defined by *FixativeMask2*.

Note that the fixative defined by *FixativeMask1* and *FixativeMask2* could possibly be multi-part fixative, i.e. 2 bits could be set in *FixativeMask1* with the fixative being a combination of both inks.

29.5 IMPLEMENTATION

A block diagram of the DNC is shown in Figure 247.

5   29.5.1   Definitions of I/O

Table 201. DNC port list and description

Port name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	System Clock.
prst_n	1	In	System reset, synchronous active low.
PCU interface			
pcu_dnc_sel	1	In	Block select from the PCU. When *pcu_dnc_sel* is high both *pcu_adr* and *pcu_dataout* are valid.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_adr[6:2]	5	In	PCU address bus. Only 5 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
dnc_pcu_rdy	1	Out	Ready signal to the PCU. When *dnc_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *dnc_pcu_datain* is valid.
dnc_pcu_datain[31:0]	32	Out	Read data bus to the PCU.
DIU interface			
dnc_diu_rreq	1	Out	DNC unit requests DRAM read. A read request must be accompanied by a valid read address.
dnc_diu_radr[21:5]	17	Out	Read address to DIU, 256-bit word aligned.
diu_dnc_rack	1	In	Acknowledge from DIU that read request has been accepted and new read address can be placed on *dnc_diu_radr*
diu_dnc_rvalid	1	In	Read data valid, active high. Indicates that valid read data is now on the read data bus, *diu_data*.
diu_data[63:0]	64	In	Read data from DIU.
HCU interface			
dnc_hcu_ready	1	Out	Indicates that DNC is ready to accept data from the HCU.
hcu_dnc_avail	1	In	Indicates valid data present on *hcu_dnc_data*.
hcu_dnc_data[5:0]	6	In	Output bi-level dot data in 6 ink planes.

DWU interface			
dwu_dnc_ready	1	In	Indicates that DWU is ready to accept data from the DNC.
dnc_dwu_avail	1	Out	Indicates valid data present on dnc_dwu_data.
dnc_dwu_data[5:0]	6	Out	Output bi-level dot data in 6 ink planes.

29.5.2   Configuration registers

The configuration registers in the DNC are programmed via the PCU interface. Refer to section 21.8.2 on page 321 for the description of the protocol and timing diagrams for reading and writing registers in the DNC. Note that since addresses in SoPEC are byte aligned and the PCU only supports 32-bit register reads and writes, the lower 2 bits of the PCU address bus are not required to decode the address space for the DNC. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of dnc_pcu_datain. Table 202 lists the configuration registers in the DNC.

Table 202. DNC configuration registers

Address (DNC_base +)	Register name	#bits	Value on reset	Description
Control registers				
0x00	Reset	1	0x1	A write to this register causes a reset of the DNC.
0x04	Go	1	0x0	Writing 1 to this register starts the DNC. Writing 0 to this register halts the DNC. When Go is asserted all counters, flags etc. are cleared or given their initial value, but configuration registers keep their values. When Go is deasserted the state-machines go to their idle states but all counters and configuration registers keep their values. This register can be read to determine if the DNC is running (1 = running, 0 = stopped).
Setup registers (constant during processing)				
0x10	MaxDot	16	0x0000	This is the maximum dot number - 1 present across a page. For example if a page contains 13824 dots, then MaxDot will be 13823.

478

				Note that this number may or may not be the same as the number of dots across the printhead as some margining may be introduced in the PHI.
0x14	LSFR	32	0x0000_0000	The current value of the LFSR register used as the 32-bit maximum length random bit generator. Users can write to this register to program a seed value for the 32-bit maximum length random bit generator. Must not be all 1s for taps implemented in XNOR form. (It is expected that writing a seed value will not occur during the operation of the LFSR). This LSFR value could also have a possible use as a random source in program code.
0x20	FixativeMask1	6	0x00	Defines the higher priority fixative plane(s). Bit 0 represents the settings for plane 0, bit 1 for plane 1 etc. For each bit: 1 = the ink plane contains fixative. 0 = the ink plane does not contain fixative.
0x24	FixativeMask2	6	0x00	Defines the lower priority fixative plane(s). Bit 0 represents the settings for plane 0, bit 1 for plane 1 etc. Used only when *FixativeMask1* planes are dead. For each bit: 1 = the ink plane contains fixative. 0 = the ink plane does not contain fixative.
0x28	FixativeRequired Mask	6	0x00	Identifies the ink planes that require fixative. Bit 0 represents the settings for plane 0, bit 1 for plane 1 etc. For each bit: 1 = the ink plane requires fixative. 0 = the ink plane does not require fixative (e.g. ink is self-fixing)

0x30	DnTableStartAdr[21:5]	17	0x0_0000	Start address of Dead Nozzle Table in DRAM, specified in 256-bit words.
0x34	DnTableEndAdr[21:5]	17	0x0_0000	End address of Dead Nozzle Table in DRAM, specified in 256-bit words, i.e. the location containing the last entry in the Dead Nozzle Table. The Dead Nozzle Table should be aligned to a 256-bit boundary, if necessary it can be padded with null entries.
0x40 - 0x54	PlaneReplacePattern[5:0]	6x6	0x00	Defines the ink replacement pattern for each of the 6 ink planes. *PlaneReplacePattern[0]* is the ink replacement pattern for plane 0, *PlaneReplacePattern[1]* is the ink replacement pattern for plane 1, etc. For each 6-bit replacement pattern for a plane, a 1 in any bit positions indicates the alternative ink planes to be used for this plane.
0x58	DiffuseEnable	6	0x3F	Defines whether, after ink replacement, error diffusion is allowed to be performed on each plane. Bit 0 represents the settings for plane 0, bit 1 for plane 1 etc. For each bit: 1 = error diffusion is enabled 0 = error diffusion is disabled
Debug registers (read only)				
0x60	DncOutputDebug	8	N/A	Bit 7 = *dwu_dnc_ready* Bit 6 = *dnc_dwu_avail* Bits 5-0 = *dnc_dwu_data*
0x64	DncReplaceDebug	14	N/A	Bit 13 = *edu_ready* Bit 12 = *iru_avail* Bits 11-6 = *iru_dn_mask* Bits 5-0 = *iru_data*
0x68	DncDiffuseDebug	14	N/A	Bit 13 = *dwu_dnc_ready* Bit 12 = *dnc_dwu_avail* Bits 11-6 = *edu_dn_mask* Bits 5-0 = *edu_data*

29.5.3   Ink replacement unit

Figure 248 shows a sub-block diagram for the ink replacement unit.

29.5.3.1 Control unit

The control unit is responsible for reading the dead nozzle table from DRAM and making it available to the DNC via the dead nozzle FIFO. The dead nozzle table is read from DRAM in single 256-bit accesses, receiving the data from the DIU over 4 clock cycles (64-bits per cycle). The protocol and timing for read accesses to DRAM is described in section 20.9.1 on page 240. Reading from DRAM is implemented by means of the state machine shown in Figure 249.

All counters and flags should be cleared after reset. When *Go* transitions from 0 to 1 all counters and flags should take their initial value. While the *Go* bit is 1, the state machine requests a read access from the dead nozzle table in DRAM provided there is enough space in its FIFO.

A modulo-4 counter, *rd_count*, is used to count each of the 64-bits received in a 256-bit read access. It is incremented whenever *diu_dnc_rvalid* is asserted. When *Go* is 1, *dn_table_radr* is set to *dn_table_start_adr*. As each 64-bit value is returned, indicated by *diu_dnc_rvalid* being asserted, *dn_table_radr* is compared to *dn_table_end_adr*.

- If *rd_count* equals 3 and *dn_table_radr* equals *dn_table_end_adr*, then *dn_table_radr* is updated to *dn_table_start_adr*.

- If *rd_count* equals 3 and *dn_table_radr* does not equal *dn_table_end_adr*, then *dn_table_radr* is incremented by 1.

A count is kept of the number of 64-bit values in the FIFO. When *diu_dnc_rvalid* is 1 data is written to the FIFO by asserting *wr_en*, and *fifo_contents* and *fifo_wr_adr* are both incremented. When *fifo_contents[3:0]* is greater than 0 and *edu_ready* is 1, *dnc_hcu_ready* is asserted to indicate that the DNC is ready to accept dots from the HCU. If *hcu_dnc_avail* is also 1 then a *dotadv* pulse is sent to the GenMask unit, indicating the DNC has accepted a dot from the HCU, and *iru_avail* is also asserted. After *Go* is set, a single *preload* pulse is sent to the GenMask unit once the FIFO contains data.

When a *rd_adv* pulse is received from the GenMask unit, *fifo_rd_adr[4:0]* is then incremented to select the next 16-bit value. If *fifo_rd_adr[1:0]* = 11 then the next 64-bit value is read from the FIFO by asserting *rd_en,* and *fifo_contents[3:0]* is decremented.

29.5.3.2 Dead nozzle FIFO

The dead nozzle FIFO conceptually is a 64-bit input, and 16-bit output FIFO to account for the 64-bit data transfers from the DIU, and the individual 16-bit entries in the dead nozzle table that are used in the GenMask unit. In reality, the FIFO is actually 8 entries deep and 64-bits wide (to accommodate two 256-bit accesses).

On the DRAM side of the FIFO the write address is 64-bit aligned while on the GenMask side the read address is 16-bit aligned, i.e. the upper 3 bits are input as the read address for the FIFO and the lower 2 bits are used to select 16 bits from the 64 bits (1st 16 bits read corresponds to bits 15-0, second 16 bits to bits 31-16 etc.).

29.5.3.3 GenMask unit

The GenMask unit generates the 6-bit *dn_mask* that is sent to the replace unit. It consists of a 10-bit delta counter and a mask register.

After *Go* is set, the GenMask unit will receive a *preload* pulse from the control unit indicating the first dead nozzle table entry is available at the output of the dead nozzle FIFO and should be loaded into the delta counter and mask register. A *rd_adv* pulse is generated so that the next dead nozzle table entry is presented at the output of the dead nozzle FIFO. The delta counter is decremented every time a *dotadv* pulse is received. When the delta counter reaches 0, it gets loaded with the current delta value output from the dead nozzle FIFO, i.e. bits 15-6, and the mask register gets loaded with mask output from the dead nozzle FIFO, i.e. bits 5-0. A *rd_adv* pulse is then generated so that the next dead nozzle table entry is presented at the output of the dead nozzle FIFO.

When the delta counter is 0 the value in the mask register is output as the *dn_mask*, otherwise the *dn_mask* is all 0s.

The GenMask unit has no knowledge of the number of dots in a line, it simply loads a counter to count the delta from one dead nozzle column to the next. Thus as described in section 29.2 on page 472 the dead nozzle table should include null identifiers if necessary so that the dead nozzle table covers the first and last nozzle column in a line.

29.5.3.4 Replace unit

Dead nozzle removal and ink replacement are implemented by the combinatorial logic shown in Figure 250. Dead nozzle removal is performed by bit-wise ANDing of the inverse of the *dn_mask* with the dot value.

The ink replacement mechanism has 6 ink replacement patterns, one per ink plane, programmable by the CPU. The dead nozzle mask is ANDed with the dot data to see if there are any planes where the dot is active but the corresponding nozzle is dead. The resultant value forms an enable, on a per ink basis, for the ink replacement process. If replacement is enabled for a particular ink, the values from the corresponding replacement pattern register are ORed into the dot data. The output of the ink replacement process is then filtered so that error diffusion is only allowed for the planes in which error diffusion is enabled.

The output of the ink replacement process is ORed with the resultant dot after dead nozzle removal. If the dot position does not contain a dead nozzle then the *dn_mask* will be all 0s and the dot, *hcu_dnc_data*, will be passed through unchanged.

29.5.4   Error Diffusion Unit

Figure 251 shows a sub-block diagram for the error diffusion unit.

29.5.4.1 Random Bit Generator

The random bit value used to arbitrarily select the direction of diffusion is generated by a maximum length 32-bit LFSR. The tap points and feedback generation are shown in Figure 252.

The LFSR generates a new bit for each dot in a line regardless of whether the dot is dead or not, i.e shifting of the LFSR is enabled when *advdot* equals 1. The LFSR can be initialised with a 32-bit programmable seed value, *random_seed*. This seed value is loaded into the LFSR whenever a write occurs to the *RandomSeed* register. Note that the seed value must not be all 1s as this causes the LFSR to lock-up.

29.5.4.2 Advance Dot Unit

The advance dot unit is responsible for determining in a given cycle whether or not the error diffuse unit will accept a dot from the ink replacement unit or make a dot available to the fixative correct unit and on to the DWU. It therefore receives the *dwu_dnc_ready* control signal from the DWU, the *iru_avail* flag from the ink replacement unit, and generates *dnc_dwu_avail* and *edu_ready* control flags.

Only the *dwu_dnc_ready* signal needs to be checked to see if a dot can be accepted and asserts *edu_ready* to indicate this. If the error diffuse unit is ready to accept a dot and the ink replacement unit has a dot available, then a *advdot* pulse is given to shift the dot into the pipeline in the diffuse unit. Note that since the error diffusion operates on 3 dots, the advance dot unit ignores *dwu_dnc_ready* initially until 3 dots have been accepted by the diffuse unit. Similarly *dnc_dwu_avail* is not asserted until the diffuse unit contains 3 dots and the ink replacement unit has a dot available.

29.5.4.3 Diffuse Unit

The diffuse unit contains the combinatorial logic to implement the truth table from Table   . The diffuse unit receives a dot consisting of 6 color planes (1 bit per plane) as well as an associated 6-bit dead nozzle mask value.

Error diffusion is applied to all 6 planes of the dot in parallel. Since error diffusion operates on 3 dots, the diffuse unit has a pipeline of 3 dots and their corresponding dead nozzle mask values. The first dot received is referred to as dot A, and the second as dot B, and the third as dot C. Dots are shifted along the pipeline whenever *advdot* is 1. A count is also kept of the number of dots received. It is incremented whenever *advdot* is 1, and wraps to 0 when it reaches *max_dot*. When the dot count is 0 dot C corresponds to the first dot in a line. When the dot count is 1 dot A corresponds to the last dot in a line.

In any given set of 3 dots only dot B can be defined as containing a dead nozzle(s). Dead nozzles are identified by bits set in *iru_dn_mask*. If dot B contains a dead nozzle(s), the corresponding bit(s) in dot A, dot C, the dead nozzle mask value for A, the dead nozzle mask value for C, the dot count, as well as the random bit value are input to the truth table logic and the dots A, B and C assigned accordingly. If dot B does not contain a dead nozzle then the dots are shifted along the pipeline unchanged.

29.5.5   Fixative Correction Unit

The fixative correction unit consists of combinatorial logic to implement fixative correction as defined in Table 203. For each output dot the DNC determines if fixative is required for the new compensated dot data word and whether fixative is activated already for that dot.

```
FixativePresent   =   ((FixativeMask1   |   FixativeMask2)  &
edu_data)  != 0
FixativeRequired = (FixativeRequiredMask & edu_data)  != 0
```

It then looks up the truth table to see what action, if any, needs to be taken.

Table 203. Truth table for fixative correction

Fixative Present	Fixative required	Action	Output
1	1	Output dot as is.	dnc_dwu_data = edu_data
1	0	Clear fixative plane.	dnc_dwu_data = (edu_data) & ~(FixativeMask1 | FixativeMask2)

0	1	Attempt to add fixative.	if (FixativeMask1 & DnMask) != 0
			  dnc_dwu_data = (edu_data) |
			(FixativeMask2 & ~DnMask)
			else
			  dnc_dwu_data = (edu_data) |
			(FixativeMask1)

0	0	Output dot as is.	dnc_dwu_data = edu_data

When attempting to add fixative the DNC first tries to add it into the plane defined by *FixativeMask1*. However, if this plane is dead then it tries to add fixative by placing it into the plane defined by *FixativeMask2*. Note that if both *FixativeMask1* and *FixativeMask2* are both all 0s then the dot data will not be changed.

30   Dotline Writer Unit (DWU)

30.1   OVERVIEW

The Dotline Writer Unit (DWU) receives 1 dot (6 bits) of color information per cycle from the DNC. Dot data received is bundled into 256-bit words and transferred to the DRAM. The DWU (in conjunction with the LLU) implements a dot line FIFO mechanism to compensate for the physical placement of nozzles in a printhead, and provides data rate smoothing to allow for local complexities in the dot data generate pipeline.

30.2   PHYSICAL REQUIREMENT IMPOSED BY THE PRINTHEAD

The physical placement of nozzles in the printhead means that in one firing sequence of all nozzles, dots will be produced over several print lines. The printhead consists of 12 rows of nozzles, one for each color of odd and even dots. Odd and even nozzles are separated by D_2 print lines and nozzles of different colors are separated by D_1 print lines. See Figure 254 for reference. The first color to be printed is the first row of nozzles encountered by the incoming paper. In the example this is color 0 odd, although is dependent on the printhead type (see [10] for other printhead arrangments). Paper passes under printhead moving downwards.

For example if the physical separation of each half row is 80μm equating to D_1=D_2=5 print lines at 1600dpi. This means that in one firing sequence, color 0 odd nozzles will fire on dotline L, color 0 even nozzles will fire on dotline L-D_1, color 1 odd nozzles will fire on dotline L-D_1-D_2 and so on over 6 color planes odd and even nozzles. The total number of lines fired over is given as 0+5+5.....+5= 0 + 11x5 =55. See Figure 255 for example diagram.

It is expected that the physical spacing of the printhead nozzles will be 80μm (or 5 dot lines), although there is no dependency on nozzle spacing. The DWU is configurable to allow other line nozzle spacings.

Table 204. Relationship between Nozzle color/sense and line firing

484

Color	Even line encountered first		Odd line encountered first	
	Sense	line	sense	line
Color 0	Even	L	even	L-5
	Odd	L-5	odd	L
Color 1	Even	L-10	even	L-15
	Odd	L-15	odd	L-10
Color 2	Even	L-20	even	L-25
	Odd	L-25	odd	L-20
Color 3	Even	L-30	even	L-35
	Odd	L-35	odd	L-30
Color 4	Even	L-40	even	L-45
	Odd	L-45	odd	L-40
Color 5	Even	L-50	even	L-55
	Odd	L-55	odd	L-50

30.3   LINE RATE DE-COUPLING

The DWU block is required to compensate for the physical spacing between lines of nozzles. It does this by storing dot lines in a FIFO (in DRAM) until such time as they are required by the LLU for dot data transfer to the printhead interface. Colors are stored separately because they are needed at different times by the LLU. The dot line store must store enough lines to compensate for the physical line separation of the printhead but can optionally store more lines to allow system level data rate variation between the read (printhead feed) and write sides (dot data generation pipeline) of the FIFOs.

A logical representation of the FIFOs is shown in Figure 256, where N is defined as the optional number of extra half lines in the dot line store for data rate de-coupling.

30.4   DOT LINE STORE STORAGE REQUIREMENTS

For an arbitrary page width of d dots (where d is even), the number of dots per half line is d/2.

For interline spacing of D_2 and inter-color spacing of D_1, with C colors of odd and even half lines, the number of half line storage is (C - 1) (D_2+D_1) + D1.

For N extra half line stores for each color odd and even, the storage is given by (N * C * 2).

The total storage requirement is ((C - 1) (D_2+D_1) + D1 + (N * C * 2)) * d/2 in bits.

Note that when determining the storage requirements for the dot line store, the number of dots per line is the page width and not necessarily the printhead width. The page width is often the dot margin number of dots less than the printhead width. They can be the same size for full bleed printing.

For example in an A4 page a line consists of 13824 dots at 1600 dpi, or 6912 dots per half dot line. To store just enough dot lines to account for an inter-line nozzle spacing of 5 dot lines it would take 55 half dot lines for color 5 odd, 50 dot lines for color 5 even and so on, giving 55+50+45...10+5+0= 330 half dot lines in total. If it is assumed that N=4 then the storage required

to store 4 extra half lines per color is 4 x 12=48, in total giving 330+48=378 half dot lines. Each half dot line is 6912 dots, at 1 bit per dot give a total storage requirement of 6912 dots x 378 half dot lines / 8 bits = Approx 319 Kbytes. Similarly for an A3 size page with 19488 dots per line, 9744 dots per half line x 378 half dot lines / 8 = Approx 899 Kbytes.

5   Table 205. Storage requirement for dot line store

Page size	Nozzle Spacing	Lines required (N=0)	Storage (N=0) Kbytes	Lines required (N=4)	Storage (N=4) Kbytes
A4	4	264	223	312	263
	5	330	278	378	319
A3	4	264	628	312	742
	5	330	785	378	899

The potential size of the dot line store makes it unfeasible to be implemented in on-chip SRAM, requiring the dot line store to be implemented in embedded DRAM. This allows a configurable dotline store where unused storage can be redistributed for use by other parts of the system.

10   30.5   NOZZLE ROW SKEW

Due to construction limitations of the bi-lithic printhead it is possible that nozzle rows may be misaligned relative to each other. Odd and even rows, and adjacent color rows may be horizontally misaligned by up to 2 dot positions. Vertical misalignment can also occur but is compensated for in the LLU and not considered here. The DWU is required to compensate for the

15   horizontal misalignment.

Dot data from the HCU (through the DNC) produces a dot of 6 colors all destined for the same physical location on paper. If the nozzle rows in the printhead are aligned as shown in Figure 254 then no adjustment of the dot data is needed.

A conceptual misaligned printhead is shown in Figure 257. The exact shape of the row alignment

20   is arbitrary, although is most likely to be sloping (if sloping, it could be sloping in either direction). The DWU is required to adjust the shape of the dot streams to take account of the join between printhead ICs. The introduction of the join shape before the data is written to the DRAM means that the PHI sees a single crossover point in the data since all lines are the same length and the crossover point (since all rows are of equal length) is a vertical line - i.e. the crossover is at the

25   same time for all even rows, and at the same time for all odd rows as shown in Figure 258.

To insert the shape of the join into the dot stream, for each line we must first insert the dots for non-printable area 1, then the printable area data (from the DNC), and then finally the dots for non-printable area 2. This can also be considered as: first produce the dots for non-printable area 1 for line n, and then a repetition of:

30   •   produce the dots for the printable area for line n (from the DNC)

     •   produce the dots for the non-printable area 2 (for line n) followed by the dots of non-printable area 1 (for line n+1)

The reason for considering the problem this way is that regardless of the shape of the join, the shape of non-printable area 2 merged with the shape of non-printable area 1 will always be a

rectangle since the widths of non-printable areas 1 and 2 are identical and the lengths of each row are identical. Hence step 2 can be accomplished by simply inserting a constant number (*MaxNozzleSkew*) of 0 dots into the stream.

For example, if the color n even row non-printable area 1 is of length X, then the length of color n even row non-printable area 2 will be of length MaxNozzleSkew - X. The split between non-printable areas 1 and 2 is defined by the *NozzleSkew* registers.

Data from the DNC is destined for the printable area only, the DWU must generate the data destined for the non-printable areas, and insert DNC dot data correctly into the dot data stream before writing dot data to the fifos. The DWU inserts the shape of the misalignment into the dot stream by delaying dot data destined to different nozzle rows by the relative misalignment skew amount.

30.6   LOCAL BUFFERING

An embedded DRAM is expected to be of the order of 256 bits wide, which results in 27 words per half line of an A4 page, and 54 words per half line of A3. This requires 27 words x 12 half colors (6 colors odd and even) = 324 x 256-bit DRAM accesses over a dotline print time, equating to 6 bits per cycle (equal to DNC generate rate of 6 bits per cycle). Each half color is required to be double buffered, while filling one buffer the other buffer is being written to DRAM. This results in 256 bits x 2 buffers x 12 half colors i.e. 6144 bits in total.

The buffer requirement can be reduced, by using 1.5 buffering, where the DWU is filling 128 bits while the remaining 256 bits are being written to DRAM. While this reduces the required buffering locally it increases the peak bandwidth requirement to the DRAM. With 2x buffering the average and peak DRAM bandwidth requirement is the same and is 6 bits per cycle, alternatively with 1.5x buffering the average DRAM bandwidth requirement is 6 bits per cycle but the peak bandwidth requirement is 12 bits per cycle. The amount of buffering used will depend on the DRAM bandwidth available to the DWU unit.

Should the DWU fail to get the required DRAM access within the specified time, the DWU will stall the DNC data generation. The DWU will issue the stall in sufficient time for the DNC to respond and still not cause a FIFO overrun. Should the stall persist for a sufficiently long time, the PHI will be starved of data and be unable to deliver data to the printhead in time. The sizing of the dotline store FIFO and internal FIFOs should be chosen so as to prevent such a stall happening.

30.7   DOTLINE DATA IN MEMORY

The dot data shift register order in the printhead is shown in Figure 254 (the transmit order is the opposite of the shift register order). In the example the type 0 printhead IC transmit order is increasing even color data followed by decreasing odd color data. The type 1 printhead IC transmit order is decreasing odd color data followed by increasing even color data. For both printhead ICs the even data is always increasing order and odd data is always decreasing. The PHI controls which printhead IC data gets shifted to.

From this it is beneficial to store even data in increasing order in DRAM and odd data in decreasing order. While this order suits the example printhead, other printheads exist where it would be beneficial to store even data in decreasing order, and odd data in increasing order,

hence the order is configurable. The order that data is stored in memory is controlled by setting the *ColorLineSense* register.

The dot order in DRAM for increasing and decreasing sense is shown in Figure 260 and Figure 261 respectively. For each line in the dot store the order is the same (although for odd lines the numbering will be different the order will remain the same). Dot data from the DNC is always received in increasing dot number order. For increasing sense dot data is bundled into 256-bit words and written in increasing order in DRAM, word 0 first, then word 1, and so on to word N, where N is the number of words in a line.

For decreasing sense dot data is also bundled into 256-bit words, but is written to DRAM in decreasing order, i.e. word N is written first then word N-1 and so on to word 0. For both increasing and decreasing sense the data is aligned to bit 0 of a word, i.e. increasing sense always starts at bit 0, decreasing sense always finishes at bit 0.

Each half color is configured independently of any other color. The *ColorBaseAdr* register specifies the position where data for a particular dotline FIFO will begin writing to. Note that for increasing sense colors the *ColorBaseAdr* register specifies the address of the first word of first line of the fifo, whereas for decreasing sense colors the *ColorBaseAdr* register specifies the address of last word of the first line of the FIFO.

Dot data received from the DNC is bundled in 256-bit words and transferred to the DRAM. Each line of data is stored consecutively in DRAM, with each line separated by *ColorLineInc* number of words.

For each line stored in DRAM the DWU increments the line count and calculates the DRAM address for the next line to store.

This process continues until *ColorFifoSize* number of lines are stored, after which the DRAM address will wrap back to the *ColorBaseAdr* address.

As each line is written to the FIFO, the DWU increments the *FifoFillLevel* register, and as the LLU reads a line from the FIFO the *FifoFillLevel* register is decremented. The LLU indicates that it has completed reading a line by a high pulse on the *llu_dwu_line_rd* line.

When the number of lines stored in the FIFO is equal to the *MaxWriteAhead* value the DWU will indicate to the DNC that it is no longer able to receive data (i.e. a stall) by deasserting the *dwu_dnc_ready* signal.

The *ColorEnable* register determines which color planes should be processed, if a plane is turned off, data is ignored for that plane and no DRAM accesses for that plane are generated.

30.8   SPECIFYING DOT FIFOS

The dot line FIFOs when accessed by the LLU are specified differently than when accessed by the DWU. The DWU uses a start address and number of lines value to specify a dot FIFO, the LLU uses a start and end address for each dot FIFO. The mechanisms differ to allow more efficient implementations in each block.

As a result of limitations in the LLU the dot FIFOs must be specified contiguously and increasing in DRAM. See section 31.6 on page 504 for further information.

30.9   IMPLEMENTATION

488

30.9.1 Definitions of I/O

Table 206. DWU I/O Definition

Port name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
DNC Interface			
dwu_dnc_ready	1	Out	Indicates that DWU is ready to accept data from the DNC.
dnc_dwu_avail	1	In	Indicates valid data present on *dnc_dwu_data*.
dnc_dwu_data[5:0]	6	In	Input bi-level dot data in 6 ink planes.
LLU Interface			
dwu_llu_line_wr	1	Out	DWU line write. Indicates that the DWU has completed a full line write. Active high
llfu_dwu_line_rd	1	In	LLU line read. Indicates that the LLU has completed a line read. Active high.
PCU Interface			
pcu_dwu_sel	1	In	Block select from the PCU. When *pcu_dwu_sel* is high both *pcu_adr* and *pcu_dataout* are valid.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_adr[7:2]	6	In	PCU address bus. Only 6 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
dwu_pcu_rdy	1	Out	Ready signal to the PCU. When *dwu_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *dwu_pcu_datain* is valid.
dwu_pcu_datain[31:0]	32	Out	Read data bus to the PCU.
DIU Interface			
dwu_diu_wreq	1	Out	DWU requests DRAM write. A write request must be accompanied by a valid write address together with valid write data and a write valid.
dwu_diu_wadr[21:5]	17	Out	Write address to DIU 17 bits wide (256-bit aligned word)
diu_dwu_wack	1	In	Acknowledge from DIU that write request has been accepted and new write address can be placed on *dwu_diu_wadr*

| dwu_diu_data[63:0] | 64 | Out | Data from DWU to DIU. 256-bit word transfer over 4 cycles

First 64-bits is bits 63:0 of 256 bit word

Second 64-bits is bits 127:64 of 256 bit word

Third 64-bits is bits 191:128 of 256 bit word

Fourth 64-bits is bits 255:192 of 256 bit word |
| dwu_diu_wvalid | 1 | Out | Signal from DWU indicating that data on *dwu_diu_data* is valid. |

30.9.2   DWU partition

30.9.3   Configuration registers

The configuration registers in the DWU are programmed via the PCU interface. Refer to section 21.8.2 on page 321 for a description of the protocol and timing diagrams for reading and writing registers in the DWU. Note that since addresses in SoPEC are byte aligned and the PCU only supports 32-bit register reads and writes, the lower 2 bits of the PCU address bus are not required to decode the address space for the DWU. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *dwu_pcu_data*. Table 207 lists the configuration registers in the DWU.

Table 207. DWU registers description

Address DWU_base+	Register	#bits	Reset	Description
Control Registers				
0x00	Reset	1	0x1	Active low synchronous reset, self de-activating. A write to this register will cause a DWU block reset.
0x04	Go	1	0x0	Active high bit indicating the DWU is programmed and ready to use. A low to high transition will cause DWU block internal states to reset (configuration registers are not reset).
Dot Line Store Configuration				
0x08 – 0x34	ColorBaseAdr[11:0][21:5]	12x17	0x000 00	Specifies the base address (in words) in memory where data from a particular half color (N) will be placed. For increasing sense colors the *ColorBaseAdr* register specifies the address of the first word of first line of the fifo, whereas for decreasing sense colors the *ColorBaseAdr* register specifies the

				address of last word of the first line of the fifo.
0x38 – 0x64	ColorFifoSize[11:0]	12x8	0x00	Indicates the number of lines in the FIFO before the line increment will wrap around in memory. Bus 0,1 - Even, Odd line color 0 Bus 2,3 - Even, Odd line color 1 Bus 4,5 - Even, Odd line color 2 Bus 6,7 - Even, Odd line color 3 Bus 8,9 - Even, Odd line color 4 Bus 10,11 - Even, Odd line color 5
0x68	ColorLineSense	2	0x2	Specifies whether data written to DRAM for this half color is increasing or decreasing sense 0 - Decreasing sense 1 - Increasing sense Bit 0 Defines even color sense, Bit 1 Defines odd color sense.
0x6C	ColorEnable	6	0x3F	Indicates whether a particular color is active or not. When inactive no data is written to DRAM for that color. 0 - Color off 1 - Color on One bit per color, bit 0 is Color 0 and so on.
0x70	MaxWriteAhead	8	0x00	Specifies the maximum number of lines that the DWU can be ahead of the LLU
0x74	LineSize	16	0x000 0	Indicates the number of dots per line produced by the DWU.
0x78	MaxNozzle Skew	4	0x0	Specifies the number of dot-pairs the DWU needs to generate to flush the data skew buffers. Corresponds to the non-printable area of the printhead.
0x7C - 0xA8	NozzleSkew	12x4	0x0	Specifies the relative skew of dot data nozzle rows in the printhead. Valid range is 0 (no skew) through to 12. Units represent dot-pairs, a skew of 1 for a row represents two dots on the

				page.
				Bus 0,1 - Even, Odd line color 0
				Bus 2,3 - Even, Odd line color 1
				Bus 4,5 - Even, Odd line color 2
				Bus 6,7 - Even, Odd line color 3
				Bus 8,9 - Even, Odd line color 4
				Bus 10,11 - Even, Odd line color 5
0xAC	ColorLineInc	8	0x00	Specifies the number of words (256-bit words) per dot line - 1.
Working Registers				
0xB0	LineDotCnt	16	0x000 0	Indicates the number of remaining dots in the current line. (Read Only)
0xB4	FifoFillLevel	8	0x00	Number of lines in the FIFO, written to but not read. (Read Only)

A low to high transition of the *Go* register causes the internal states of the DWU to be reset. All configuration registers will remain the same. The block indicates the transition to other blocks via the *dwu_go_pulse* signal.

30.9.4   Data skew

5   The data skew block inserts the shape of the printhead join into the dot data stream by delaying dot data by the relative nozzle skew amount (given by *nozzle_skew*). It generates zero fill data introduced introduced into the dot data stream to achieve the relative skew (and also to flush dot data from the delay registers).

The data skew block consists of 12 12-bit shift registers, one per color odd and even. The shift

10   registers are in groups of 6, one group for even colors, and one for odd colors. Each time a valid data word is received from the DNC the dot data is shifted into either the odd or even group of shift registers. The *odd_even_sel* register determines which group of shift registers are valid for that cycle and alternates for each new valid data word. When a valid word is received for a group of shift registers, the shift register is shifted by one location with the new data word shifted into the

15   registers (the top word in the register will be discarded).

When the dot counter determines that the data skew block should zero fill (*zero_fill*), the data skew block will shift zero dot data into the shift registers until the line has completed. During this time the DNC will be stalled by the de-assertion of the *dwu_dnc_ready* signal.

The data skew block selects dot data from the shift registers and is passed to the buffer address

20   generator block. The data bits selected is determined by the configured index values in the *NozzleSkew* registers.

```
// determine when data is valid
data_valid = (((dnc_dwu_avail == 1)OR(zero_fill == 1)) AND
(dwu_ready ==1))
```

25
```
// implement the zero fill mux
if (zero_fill == 1) then
   dot_data_in = 0
```

492

```
          else
             dot_data_in = dnc_dwu_data
          // the data delay buffers
          if (dwu_go_pulse ==1) then
             data_delay[1:0][11:0][5:0]    = 0                  // reset all
          delay buffer odd=1,even=0
             odd_even_sel                  = 0
          elsif (data_valid == 1) then {
             odd_even_sel = ~odd_even_sel
             // update the odd/even buffers, with shift
             data_delay[odd_even_sel][11:1][5:0]=
          data_delay[odd_even_sel][10:0][5:0] // shift data
             data_delay[odd_even_sel][0][5:0]    =    dot_data_in[5:0]
          // shift in new data
             // select the correct output data
             for (i=0;i<6; i++) {
                // skew selector
                skew       =        nozzle_skew[      {i,odd_even_sel}      ]
          // temporary variable
                // data select array, include data delay and input dot
          data
                data_select[12:0]   =    {data_delay[odd_even_sel][11:0],
          dot_data_in}
                // mux output the data word to next block (13 to 1 mux)
                dot_data[i] = data_select[skew][i]
                }
             }
```

30.9.5   Fifo fill level

The DWU keeps a running total of the number of lines in the dot store FIFO. Each time the DWU writes a line to DRAM (determined by the DIU interface subblock and signalled via *line_wr*) it increments the *filllevel* and signals the line increment to the LLU (pulse on *dwu_llu_line_wr*). Conversely if it receives an active *llu_dwu_line_rd* pulse from the LLU, the *filllevel* is decremented. If the *filllevel* increases to the programmed max level (*max_write_ahead*) then the DWU stalls and indicates back to the DNC by de-asserting the *dwu_dnc_ready* signal.

If one or more of the DIU buffers fill, the DIU interface signals the fill level logic via the *buf_full* signal which in turn causes the DWU to de-assert the *dwu_dnc_ready* signal to stall the DNC. The *buf_full* signals will remain active until the DIU services a pending request from the full buffer, reducing the buffer level.

When the dot counter block detects that it needs to insert zero fill dots (*zero_fill* equals 1) the DWU will   stall the DNC while the zero dots are being generated (by de-asserting *dwu_dnc_ready)*, but will allow the data skew block to generate zero fill data (the *dwu_ready* signal).

```
dwu_dnc_ready    =   ~((buf_full==  1)   OR   (filllevel   ==
max_write_ahead ) OR (zero_fill == 1))
dwu_ready             =   ~((buf_full==  1)   OR   (filllevel   ==
max_write_ahead ))
```

5   The DWU does not increment the fill level until a complete line of dot data is in DRAM not just a complete line received from the DNC. This ensures that the LLU cannot start reading a partial line from DRAM before the DWU has finished writing the line.

The fill level is reset to zero each time a new page is started, on receiving a pulse via the *dwu_go_pulse* signal.

10   The line fifo fill level can be read by the CPU via the PCU at any time by accessing the *FifoFillLevel* register.

30.9.6   Buffer address generator

30.9.6.1 Buffer address generator description

The buffer address generator subblock is responsible for accepting data from the data skew block

15   and writing it to the DIU buffers in the correct order.

The buffer address and active bit-write for a particular dot data write is calculated by the buffer address generator based on the dot count of the current line, programmed sense of the color and the line size.

All configuration registers should be programmed while the *Go* bit is set to zero, once complete

20   the block can be enabled by setting the *Go* bit to one. The transition from zero to one will cause the internal states to reset.

If the *color_line_sense* signal for a color is one (i.e. increasing) then the bit-write generation is straight forward as dot data is aligned with a 256-bit boundary. So for the first dot in that color, the bit 0 of the *wr_bit* bus will be active (in buffer word 0), for the second dot bit 1 is active and so on

25   to the 255th dot where bit 63 is active (in buffer word 3). This is repeated for all 256-bit words until the final word where only a partial number of bits are written before the word is transferred to DRAM.

If *color_line_sense* signal for a color is zero (i.e. decreasing) the bit-write generation for that color is adjusted by an offset calculated from the pre-programmed line length (*line_size*). The offset

30   adjusts the bit write to allow the line to finish on a 256-bit boundary. For example if the line length was 400, for the first dot received bit 7 (line length is halved because of odd/even lines of color) of the *wr_bit* is active (buffer word 3), the second bit 6 (buffer word 3), to the 200th dot of data with bit 0 of *wr_bit* active (buffer word 0).

30.9.6.2 Bit-write decode

35   The buffer address generator contains 2 instances of the bit-write decode, one configured for odd dot data the other for even. The counter (either up or down counter) used to generate the addresses is selected by the *color_line_sense* signal. Each block determines if it is active on this cycle by comparing its configured type with the current dot count address and the *data_active* signal.

40   The *wr_bit* bus is a direct decoding of the lower 6 count bits (*count[6:1]*), and the DIU buffer address is the remaining higher bits of the counter (*count[10:7]*).

The signal generation is given as follows:

```
                    // determine the counter to use
                    if (color_line_sense == 1 )
                      count = up_cnt[10:0]
                    else
                      count = dn_cnt[10:0]
                    // determine if active, based on instance type
                    wr_en            = data_active & (count[0] ^ odd_even_type)
                    // odd =1, even =0
                    // determine the bit write value
                    wr_bit[63:0]   = decode(count[6:1])
                    // determine the buffer 64-bit address
                    wr_adr[3:0]    = count[10:7]
```

30.9.6.3 Up counter generator

The up counter increments for each new dot and is used to determine the write position of the dot in the DIU buffers for increasing sense data. At the end of each line of dot data (as indicated by *line_fin*), the counter is rounded up to the nearest 256-bit word boundary. This causes the DIU buffers to be flushed to DRAM including any partially filled 256-bit words. The counter is reset to zero if the *dwu_go_pulse* is one.

```
                    // Up-Counter Logic
                    if (dwu_go_pulse == 1) then {
                      up_cnt[10:0] = 0
                    elsif (line_fin == 1 ) then
                      // round up
                      if (up_cnt[8:1] != 0)
                        up_cnt[10:9]++
                      else
                        up_cnt[10:9]
                      // bit-selector
                      up_cnt[7:0]=0
                    elsif (data_valid == 1) then
                      up_cnt[7:0]++
```

30.9.6.4 Down counter generator

The down counter logic decrements for each new dot and is used to determine the write position of the dot in the DUI buffers for decreasing sense data. When the *dwu_go_pulse* bit is one the lower bits (i.e. 8 to 0) of the counter are reset to line size value (*line_size*), and the higher bits to zero. The bits used to determine the bit-write values and 64-bit word addresses in the DIU buffers begin at line size and count down to zero. The remaining higher bits are used to determine the DIU buffer 256-bit address and buffer fill level, begin at zero and count up. The counter is active when valid dot data is present, i.e. *data_valid* equals 1.

When the end of line is detected (*line_fin* equals 1) the counter is rounded to the next 256-bit word, and the lower bits are reset to the line size value.

495

```
//Down-Counter Logic
if (dwu_go_pulse == 1) then
   dn_cnt[8:0]  = line_size[8:0]
   dn_cnt[10:9] = 0
elsif (line_fin == 1 ) then
   // perform rounding up
   if (dn_cnt[8:1] != 0)
      dn_cnt[10:9]++
   else
      dn_cnt[10:9]
   // bit-select is reset
   dn_cnt[8:0]=line_size[8:0]    // bit select bits
elsif (data_valid == 1) then
   dn_cnt[8:0] --
   dn_cnt[10:9]++
```

30.9.6.5 Dot counter

The dot counter simply counts each active dot received from the data skew block. It sets the counter to *line_size* and decrements each time a valid dot is received. When the count equals zero the *line_fin* signal is pulsed and the counter is reset to *line_size*.

When the count is less than the *max_nozzle_skew* * 2 value the dot counter indicates to the data skew block to zero fill the remainder of the line (via the *zero_fill* signal). Note that the *max_nozzle_skew* units are dot-pairs as opposed to dots, hence the by 2 multiplication for comparison with the dot counter.

The counter is reset to *line_size* when *dwu_go_pulse* is 1.

30.9.7   DIU buffer

The DIU buffer is a 64 bit x 8 word dual port register array with bit write capability. The buffer could be implemented with flip-flops should it prove more efficient.

30.9.8   DIU interface

30.9.8.1 DIU interface general description

The DIU interface determines when a buffer needs a data word to be transferred to DRAM. It generates the DRAM address based on the dot line position, the color base address and the other programmed parameters. A write request is made to DRAM and when acknowledged a 256-bit data word is transferred. The interface determines if further words need to be transferred and repeats the transfer process.

If the FIFO in DRAM has reached its maximum level, or one of the buffers has temporarily filled, the DWU will stall data generation from the DNC.

A similar process is repeated for each line until the end of page is reached. At the end of a page the CPU is required to reset the internal state of the block before the next page can be printed. A low to high transition of the *Go* register will cause the internal block reset, which causes all registers in the block to reset with the exception of the configuration registers. The transition is indicated to subblocks by a pulse on *dwu_go_pulse* signal.

30.9.8.2 Interface controller

The interface controller state machine waits in *Idle* state until an active request is indicated by the read pointer (via the *req_active* signal). When an active request is received the machine proceeds to the *ColorSelect* state to determine which buffers need a data transfer. In the *ColorSelect* state it cycles through each color and determines if the color is enabled (and consequently the buffer needs servicing), if enabled it jumps to the *Request* state, otherwise the *color_cnt* is incremented and the next color is checked.

In the *Request* state the machine issues a write request to the DIU and waits in the *Request* state until the write request is acknowledged by the DIU (*diu_dwu_wack*). Once an acknowledge is received the state machine clocks through 4 cycles transferring 64-bit data words each cycle and incrementing the corresponding buffer read address. After transferring the data to the DIU the machine returns to the *ColorSelect* state to determine if further buffers need servicing. On the transition the controller indicates to the address generator (*adr_update*) to update the address for that selected color.

If all colors are transferred (*color_cnt* equal to 6) the state machine returns to *Idle*, updating the last word flags (*group_fin*) and request logic (*req_update*).

The *dwu_diu_wvalid* signal is a delayed version of the *buf_rd_en* signal to allow for pipeline delays between data leaving the buffer and being clocked through to the DIU block.

The state machine will return from any state to *Idle* if the reset or the *dwu_go_pulse* is 1.

30.9.8.3 Address generator

The address generator block maintains 12 pointers (*color_adr[11:0]*) to DRAM corresponding to current write address in the dot line store for each half color. When a DRAM transfer occurs the address pointer is used first and then updated for the next transfer for that color. The pointer used is selected by the *req_sel* bus, and the pointer update is initiated by the *adr_update* signal from the interface controller.

The pointer update is dependent on the sense of the color of that pointer, the pointer position in a line and the line position in the FIFO. The programming of the *color_base_adr* needs to be adjusted depending of the sense of the colors. For increasing sense colors the *color_base_adr* specifies the address of the first word of first line of the fifo, whereas for decreasing sense colors the *color_base_adr* specifies the address of last word of the first line of the FIFO.

For increasing colors, the initialization value (i.e. when *dwu_go_pulse* is 1) is the *color_base_adr*. For each word that is written to DRAM the pointer is incremented. If the word is the last word in a line (as indicated by *last_wd* from that read pointers) the pointer is also incremented. If the word is the last word in a line, and the line is the last line in the FIFO (indicated by *fifo_end* from the line counter) the pointer is reset to *color_base_adr*.

In the case of decreasing sense colors, the initialization value (i.e. when *dwu_go_pulse* is 1) is the *color_base_adr*. For each line of decreasing sense color data the pointer starts at the line end and decrements to the line start. For each word that is written to DRAM the pointer is decremented. If the word is the last word in a line the pointer is incremented by *color_line_inc* * 2 + 1. One line length to account for the line of data just written, and another line length for the next line to be

written. If the word is the last word in a line, and the line is the last line in the FIFO the pointer is reset to the initialization value (i.e. *color_base_adr).*

The address is calculated as follows:

```
5       if (dwu_go_pulse == 1) then
            color_adr[11:0] = color_base_adr[11:0][21:5]
        elsif (adr_update == 1) then {
            // determine the color
            color = req_sel[3:0]
10          // line end and fifo wrap
            if ((fifo_end[color] == 1) AND (last_wd == 1)) then {
                // line end and fifo wrap
                color_adr[color] = color_base_adr[color][21:5]
            }
15          elsif ( last_wd == 1) then {
                // just a line end no fifo wrap
                if (color_line_sense[color  %  2]  ==  1)  then     //
            increasing sense
                    color_adr[color] ++
20              else                                    // decreasing
            sense
                    color_adr[color]     =     color_adr[color]    +    (
            color_line_inc * 2) + 1
            }
25          else {
                // regular word write
                if (color_line_sense[color  %  2]  ==  1)  then     //
            increasing sense
                    color_adr[color]++
30              else                                    // decreasing sense
                    color_adr[color]--
            }
        }
        // select the correct address, for this transfer
35      dwu_diu_wadr = color_adr[req_sel]
```

30.9.8.4 Line count

The line counter logic counts the number of dot data lines stored in DRAM for each color. A separate pointer is maintained for each color. A line pointer is updated each time the final word of a line is transferred to DRAM. This is determined by a combination of *adr_update* and *last_wd*

40 signals. The pointer to update is indicated by the *req_sel* bus.

When an update occurs to a pointer it is compared to zero, if it is non-zero the count is decremented, otherwise the counter is reset to *color_fifo_size.* If a counter is zero the *fifo_end* signals is set high to indicates to the address generator block that the line is the last line of this colors fifo.

If the *dwu_go_pulse* signal is one the counters are reset to *color_fifo_size*.

```
if (dwu_go_pulse == 1) then
    line_cnt[11:0] = color_fifo_size[11:0]
elsif ((adr_update == 1) AND (last_wd == 1)) then {
    // determine the pointer to operate on
    color = req_sel[3:0]
    // update the pointer
    if (line_cnt[color] == 0) then
        line_cnt[color] = color_fifo_size[color]
    else
        line_cnt[i] --
    }
    // count is zero its the last line of fifo
    for(i=0 ;i <12;i++){
        fifo_end[i] = (line_cnt[i] == 0)
    }
```

30.9.8.5 Read Pointer

The read pointer logic maintains the buffer read address pointers. The read pointer is used to determine which 64-bit words to read from the buffer for transfer to DRAM.

The read pointer logic compares the read and write pointers of each DIU buffer to determine which buffers require data to be transferred to DRAM, and which buffers are full (the *buf_full* signal).

Buffers are grouped into odd and even buffers groups. If an odd buffer requires DRAM access the *odd_pend* signals will be active, if an even buffer requires DRAM access the *even_pend* signals will be active. If both odd and even buffers require DRAM access at exactly the same time, the even buffers will get serviced first. If a group of odd buffers are being serviced and an even buffer becomes pending, the odd group of buffers will be completed before the starting the even group, and vice versa.

If any buffer requires a DRAM transfer, the logic will indicate to the interface controller via the *req_active* signal, with the *odd_even_sel* signal determining which group of buffers get serviced. The interface controller will check the *color_enable* signal and issue DRAM transfers for all enabled colors in a group. When the transfers are complete it tells the read pointer logic to update the requests pending via *req_update* signal.

The *req_sel[3:0]* signal tells the address generator which buffer is being serviced, it is constructed from the *odd_even_sel* signal and the *color_cnt[2:0]* bus from the interface controller. When data is being transferred to DRAM the word pointer and read pointer for the corresponding buffer are updated. The *req_sel* determines which pointer should be incremented.

```
    // determine if request is active even
    if ( wr_adr[0][3:2] != rd_adr[0][3:2] )
        even_pend = 1
    else
        even_pend = 0
```

499

```
                    // determine if request is active odd
                    if ( wr_adr[1][3:2] != rd_adr[1][3:2] )
                      even_pend = 1
                    else
 5                    even_pend = 0
                    // determine if any buffer is full
                    if ((wr_adr[0][3:0] - rd_adr[0][3:0]) > 7)OR((wr_adr[1][3:0]
                    - rd_adr[1][3:0])> 7)) then
                      buf_full = 1
10                  //  fixed  servicing  order,  only  update  when  controller
                    dictates so
                    if (req_update == 1) then {
                      if (even_pend == 1) then        // even always first
                        odd_even_sel = 0
15                      req_active   = 1
                      elsif (odd_pend == 1 ) then      // then check odd
                        odd_even_sel = 0
                        req_active   = 1
                      else                            // nothing active
20                      odd_even_sel = 0
                        req_active   = 0
                    }
                    // selected requestor
                    req_sel[3:0]  =  {color_cnt[2:0]  ,  odd_even_sel}     //
25                  concatentation
```

The read address pointer logic consists of 2 2-bit counters and a word select pointer. The pointers are reset when *dwu_go_pulse* is one. The word pointer (*word_ptr*) is common to all buffers and is used to read out the 64-bit words from the DIU buffer. It is incremented when *buf_rd_en* is active. When a group of buffers are updated the state machine increments the read pointer

30 (*rd_ptr[odd_even_sel]*) via the *group_fin* signal. A concatenation of the read pointer and the word pointer are use to construct the buffer read address. The read pointers are not reset at the end of each line.

```
                    // determine which pointer to update
                    if (dwu_go_pulse == 1) then
35                    rd_ptr[1:0]  = 0
                      word_ptr     = 0
                    elsif (buf_rd_en == 1) then {
                      word_ptr++                          // word pointer update
                    elsif (group_fin == 1) then
40                    rd_ptr[odd_even_sel]++              // update the read
                    pointer
                    // create the address from the pointer,and word reader
                    rd_adr[odd_even_sel] = {rd_ptr[odd_even_sel],word_ptr}    //
                    concatenation
```

The read pointer block determines if the word being read from the DIU buffers is the last word of a line. The buffer address generator indicate the last dot is being written into the buffers via the *line_fin* signal. When received the logic marks the 256-bit word in the buffers as the last word. When the last word is read from the DIU buffer and transferred to DRAM, the flag for that word is reflected to the address generator.

```
// line end set the flags
if (dwu_go_pulse == 1) then
    last_flag[1:0][1:0] = 0
elsif (line_fin == 1 ) then
    // determines the current 256-bit word even been written
    to
    last_flag[0][wr_adr[0][2]] = 1    // even group.flag
    // determines the current 256-bit word odd been written to
    last_flag[1][wr_adr[1][2]] = 1    // odd group flag
    // last word reflection to address generator
    last_wd = last_flag[odd_even_sel][rd_ptr[req_sel][0]]
    // clear the flag
    if (group_fin == 1 ) then
        last_flag[odd_even_sel][rd_ptr[req_sel][0]] = 0
```

When a complete line has been written into the DIU buffers (but has not yet been transferred to DRAM), the buffer address generator block will pulse the *line_fin* signal. The DWU must wait until all enabled buffers are transferred to DRAM before signaling the LLU that a complete line is available in the dot line store (*dwu_llu_line_wr* signal). When the *line_fin* is received all buffers will require transfer to DRAM. Due to the arbitration, the even group will get serviced first then the odd. As a result the line finish pulse to the LLU is generated from the *last_flag* of the odd group.

```
// must be odd,odd group transfer complete and the last word
dwu_llu_line_wr = odd_even_sel AND group_fin AND last_wd
```

31   Line Loader Unit (LLU)

31.1   OVERVIEW

The Line Loader Unit (LLU) reads dot data from the line buffers in DRAM and structures the data into even and odd dot channels destined for the same print time. The blocks of dot data are transferred to the PHI and then to the printhead. Figure 267 shows a high level data flow diagram of the LLU in context.

31.2   PHYSICAL REQUIREMENT IMPOSED BY THE PRINTHEAD

The DWU re-orders dot data into 12 separate dot data line FIFOs in the DRAM. Each FIFO corresponds to 6 colors of odd and even data. The LLU reads the dot data line FIFOs and sends the data to the printhead interface. The LLU decides when data should be read from the dot data line FIFOs to correspond with the time that the particular nozzle on the printhead is passing the current line. The interaction of the DWU and LLU with the dot line FIFOs compensates for the physical spread of nozzles firing over several lines at once. For further explanation see Section 30 Dotline Writer Unit (DWU) and Section 32 PrintHead Interface (PHI). Figure 268 shows the

physical relationship of nozzle rows and the line time the LLU starts reading from the dot line store.

Within each line of dot data the LLU is required to generate an even and odd dot data stream to the PHI block. Figure 269 shows the even and dot streams as they would map to an example bi-lithic printhead. The PHI block determines which stream should be directed to which printhead IC.

31.3   DOT GENERATE AND TRANSMIT ORDER

The structure of the printhead ICs dictate the dot transmit order to each printhead IC. The LLU reads data from the dot line FIFO, generates an even and odd dot stream which is then re-ordered (in the PHI) into the transmit order for transfer to the printhead.

The DWU separates dot data into even and odd half lines for each color and stores them in DRAM. It can store odd or even dot data in increasing or decreasing order in DRAM. The order is programmable but for descriptive purposes assume even in increasing order and odd in decreasing order. The dot order structure in DRAM is shown in Figure 261.

The LLU contains 2 dot generator units. Each dot generator reads dot data from DRAM and generates a stream of odd or even dots. The dot order may be increasing or decreasing depending on how the DWU was programmed to write data to DRAM. An example of the even and odd dot data streams to DRAM is shown in Figure 270. In the example the odd dot generator is configured to produce odd dot data in decreasing order and the even dot generator produces dot data in increasing order.

The PHI block accepts the even and odd dot data streams and reconstructs the streams into transmit order to the printhead.

The LLU line size refers to the page width in dots and not necessarily the printhead width. The page width is often the dot margin number of dots less than the printhead width. They can be the same size for full bleed printing.

31.4   LLU START-UP

At the start of a page the LLU must wait for the dot line store in DRAM to fill to a configured level (given by *FifoReadThreshold)* before starting to read dot data. Once the LLU starts processing dot data for a page it must continue until the end of a page, the DWU (and other PEP blocks in the pipeline) must ensure there is always data in the dot line store for the LLU to read, otherwise the LLU will stall, causing the PHI to stall and potentially generate a print error. The *FifoReadThreshold* should be chosen to allow for data rate mismatches between the DWU write side and the LLU read side of the dot line FIFO. The LLU will not generate any dot data until *FifoReadThreshold* level in the dot line FIFO is reached.

Once the *FifoReadThreshold* is reached the LLU begins page processing, the *FifoReadThreshold* is ignored from then on.

When the LLU begins page processing it produces dot data for all colors (although some dot data color may be null data). The LLU compares the line count of the current page, when the line count exceeds the *ColorRelLine* configured value for a particular color the LLU will start reading from that colors FIFO in DRAM. For colors that have not exceeded the *ColorRelLine* value the LLU will

generate null data (zero data) and not read from DRAM for that color. *ColorRelLine[N]* specifies the number of lines separating the Nth half color and the first half color to print on that page. For the example printhead shown in Figure 268, color 0 odd will start at line 0, the remaining colors will all have null data. Color 0 odd will continue with real data until line 5, when color 0 odd and even will contain real data the remaining colors will contain null data. At line 10, color 0 odd and even and color 1 odd will contain real data, with remaining colors containing null data. Every 5 lines a new half color will contain real data and the remaining half colors null data until line 55, when all colors will contain real data. In the example *ColorRelLine[0]* =5, *ColorRelLine[1]* =0, *ColorRelLine[2]* =15, *ColorRelLine[3]* =10 .. etc.

It is possible to turn off any one of the color planes of data (via the *ColorEnable* register), in such cases the LLU will generate zeroed dot data information to the PHI as normal but will not read data from the DRAM.

31.4.1 LLU bandwidth requirements

The LLU is required to generate data for feeding to the printhead interface, the rate required is dependent on the printhead construction and on the line rate configured. The maximum data rate the LLU can produce is 12 bits of dot data per cycle, but the PHI consumes at 12 bits every 2 *pclk* cycles out of 3, i.e. 8 bits per *pclk* cycle. Therefore the DRAM bandwidth requirement for a double buffered LLU is 8 bits per cycle on average. If 1.5 buffering is used then the peak bandwidth requirement is doubled to 16 bits per cycle but the average remains at 8 bits per cycle. Note that while the LLU and PHI could produce data at the 8 bits per cycle rate, the DWU can only produce data at 6 bits per cycle rate.

31.5 VERTICAL ROW SKEW

Due to construction limitations of the bi-lithic printhead it is possible that nozzle rows may be misaligned relative to each other. Odd and even rows, and adjacent color rows may be horizontally misaligned by up to 2 dot positions. Vertical misalignment can also occur between both printhead ICs used to construct the printhead. The DWU compensates for the horizontal misalignment (see Section 30.5), and the LLU compensates for the vertical misalignment.

For each color odd and even the LLU maintains 2 pointers into DRAM, one for feeding printhead A (*CurrentPtrA*) and other for feeding printhead B (*CurrentPtrB*). Both pointers are updated and incremented in exactly the same way, but differ in their initial value programming. They differ by vertical skew number of lines, but point to the same relative position within a line.

At the start of a line the LLU reads from the FIFO using *CurrentPtrA* until the join point between the printhead ICs is reached (specified by *JoinPoint*), after which the LLU reads from DRAM using *CurrentPtrB*. If the *JoinPoint* coincides with a 256-bit word boundary, the swap over from pointer A to pointer B is straightforward. If the *JoinPoint* is not on a 256-bit word boundary, the LLU must read the 256-bit word of data from *CurrentPtrA* location, generate the dot data up to the join point and then read the 256-bit word of data from *CurrentPtrB* location and generate dot data from the join point to the word end. This means that if the *JoinPoint* is not on a 256-bit boundary then the LLU is required to perform an extra read from DRAM at the join point and not increment the address pointers.

31.5.1 Dot line FIFO initialization

For each dot line FIFO there are 2 pointers reading from it, each skewed by a number of dot lines in relation to the other (the skew amount could be positive or negative). Determining the exact number of valid lines in the dot line store is complicated by two pointers reading from different positions in the FIFO. It is convenient to remove the problem by pre-zeroing the dot line FIFOs effectively removing the need to determine exact data validity. The dot FIFOs can be initialized in a number of ways, including

- the CPU writing 0s,
- the LBD/SFU writing a set of 0 lines (16 bits per cycle),
- the HCU/DNC/DWU being programmed to produce 0 data

31.6 SPECIFYING DOT FIFOS

The dot line FIFOs when accessed by the LLU are specified differently than when accessed by the DWU. The DWU uses a start address and number of lines value to specify a dot FIFO, the LLU uses a start and end address for each dot FIFO. The mechanisms differ to allow more efficient implementations in each block.

The start address for each half color N is specified by the *ColorBaseAdr[N]* registers and the end address (actually the end address plus 1) is specified by the *ColorBaseAdr[N+1]*. Note there are 12 colors in total, 0 to 11, the *ColorBaseAdr[12]* register specifies the end of the color 11 dot FIFO and not the start of a new dot FIFO. As a result the dot FIFOs must be specified contiguously and increasing in DRAM.

31.7 IMPLEMENTATION

31.7.1 LLU partition

31.7.2 Definitions of I/O

Table 208. LLU I/O definition

Port name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	System clock
prst_n	1	In	System reset, synchronous active low
PHI Interface			
llu_phi_data[1:0][5:0]	2x6	Out	Dot Data from LLU to the PHI, each bit is a color plane 5 downto 0. Bus 0 - Even dot data stream Bus 1 - Odd dot data stream Data is active when corresponding bit is active in *llu_phi_avail* bus
phi_llu_ready[1:0]	2	In	Indicates that PHI is ready to accept data from the LLU 0 - Even dot data stream

			1 - Odd dot data stream
llu_phi_avail[1:0]	2	Out	Indicates valid data present on corresponding *llu_phi_data*. 0 - Even dot data stream 1 - Odd dot data stream
DIU Interface			
llu_diu_rreq	1	Out	LLU requests DRAM read. A read request must be accompanied by a valid read address.
llu_diu_radr[21:5]	17	Out	Read address to DIU 17 bits wide (256-bit aligned word).
diu_llu_rack	1	In	Acknowledge from DIU that read request has been accepted and new read address can be placed on *llu_diu_radr*
diu_data[63:0]	64	In	Data from DIU to LLU. Each access is 256-bits received over 4 clock cycles First 64-bits is bits 63:0 of 256 bit word Second 64-bits is bits 127:64 of 256 bit word Third 64-bits is bits 191:128 of 256 bit word Fourth 64-bits is bits 255:192 of 256 bit word
diu_llu_rvalid	1	In	Signal from DIU telling LLU that valid read data is on the *diu_data* bus
DWU Interface			
dwu_llu_line_wr	1	In	DWU line write. Indicates that the DWU has completed a full line write. Active high
llu_dwu_line_rd	1	Out	LLU line read. Indicates that the LLU has completed a line read. Active high.
PCU Interface			
pcu_llu_sel	1	In	Block select from the PCU. When *pcu_llu_sel* is high both *pcu_adr* and *pcu_dataout* are valid.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_adr[7:2]	6	In	PCU address bus. Only 6 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
llu_pcu_rdy	1	Out	Ready signal to the PCU. When *llu_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *llu_pcu_datain* is valid.
llu_pcu_datain[31:0]	32	Out	Read data bus to the PCU.

31.7.3   Configuration registers

The configuration registers in the LLU are programmed via the PCU interface. Refer to section 21.8.2 on page 321 for a description of the protocol and timing diagrams for reading and writing registers in the LLU. Note that since addresses in SoPEC are byte aligned and the PCU only supports 32-bit register reads and writes, the lower 2 bits of the PCU address bus are not required to decode the address space for the LLU. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *llu_pcu_datain*. Table 209 lists the configuration registers in the LLU.

Table 209. LLU registers description

Address LLU_base +	Register	#bits	Reset	Description
Control Registers				
0x00	Reset	1	0x1	Active low synchronous reset, self de-activating. A write to this register will cause a LLU block reset.
0x04	Go	1	0x0	Active high bit indicating the LLU is programmed and ready to use. A low to high transition will cause LLU block internal states to reset.
Configuration				
0x08 - 0x38	ColorBaseAdr[12:0][21:5]	13x17	0x000 00	Specifies the base address (in words) in memory where data from a particular half color (N) will be placed. Also specifies the end address + 1 (256-bit words) in memory where fifo data for a particular half color ends. For color N the start address is ColorBaseAdr[N] and the end address +1 is ColorBase-Adr[N+1]
0x3C	ColorEnable	6	0x3F	Indicates whether a particular color is active or not. When inactive no data is written to DRAM for that color. 0 - Color off 1 - Color on One bit per color, bit 0 is Color 0 and so on.
0x40	LineSize	16	0x000 0	Indicates the number of dots per line.

0x44	FifoReadThreshold	8	0x00	Specifies the number of lines that should be in the FIFO before the LLU starts reading.
0x48 - 0x74	ColorRelLine[11:0]	12x8	0x00	Specifies the relative number of lines to wait from the first before starting to read dot data from the corresponding dot data FIFO Bus 0,1 - Even, Odd line color 0 Bus 2,3 - Even, Odd line color 1 Bus 4,5 - Even, Odd line color 2 Bus 6,7 - Even, Odd line color 3 Bus 8,9 - Even, Odd line color 4 Bus 10,11 - Even, Odd line color 5
0x78 - 0x7C	JoinPoint	2x16	0x000 0	Specifies the join point in dots between both printhead ICs. Bus 0 - Even dot generator join point Bus 1 - Odd dot generator join point
0x80 - 0x84	JoinWord	2x8	0x00	Specifies the join point in words between both printhead ICs. Bus 0 - Even dot generator join point Bus 1 - Odd dot generator join point
0x90-0xBC	CurrentAdrA[11:0][2 1:5]	12x17	0x000 0	Current Address pointers associated with printhead A Bus 0,1 - Even, Odd line color 0 Bus 2,3 - Even, Odd line color 1 Bus 4,5 - Even, Odd line color 2 Bus 6,7 - Even, Odd line color 3 Bus 8,9 - Even, Odd line color 4 Bus 10,11 - Even, Odd line color 5 Working registers
0xC0 0xEC	-CurrentAdrB[11:0][2 1:5]	12x17	0x000 0	Current Address pointers associated with printhead B Bus 0,1 - Even, Odd line color 0 Bus 2,3 - Even, Odd line color 1 Bus 4,5 - Even, Odd line color 2 Bus 6,7 - Even, Odd line color 3 Bus 8,9 - Even, Odd line color 4 Bus 10,11 - Even, Odd line color 5 Working registers

Working Registers				
0xF0	FifoFillLevel	8	0x00	Number of lines in the dot line FIFO, line written in but not read out. (Read Only)

A low to high transition of the *Go* register causes the internal states of the LLU to be reset. All configuration registers will remain the same. The block indicates the transition to other blocks via the *llu_go_pulse* signal.

31.7.4 Dot generator

The dot generator block is responsible for reading dot data from the DIU buffers and sending the dot data in the correct order to the PHI block. The dot generator waits for *llu_en* signal from the fifo fill level block, once active it starts reading data from the 6 DIU buffers and generating dot data for feeding to the PHI.

In the LLU there are two instances of the dot generator, one generating odd data and the other generating even data.

At any time the ready bit from the PHI could be de-asserted, if this happens the dot generator will stop generating data, and wait for the ready bit to be re-asserted.

31.7.4.1 Dot count

In normal operation the dot counter will wait for the *llu_en* and the ready to be active before starting to count. The dot count will produce data as long as the *phi_llu_ready* is active. If the *phi_llu_ready* signal goes low the count will be stalled.

The dot counter increments for each dot that is processed per line. It is used to determine the line finish position, and the bit select value for reading from the DIU buffers. The counter is reset after each line is processed (*line_fin* signal). It determines when a line is finished by comparing the dot count with the configured line size divided by 2 (note that odd numbers of dots will be rounded down).

```
// define the line finish
if (dot_cnt[14:0] == line_size[15:1] )then
   line_fin = 1
else
   line_fin = 0
// determine if word is valid
dot_active = ((llu_en == 1) AND (phi_llu_ready == 1) AND
(buf_emp == 0))
// counter logic
if (llu_go_pulse == 1) then
   dot_cnt = 0
elsif ((dot_active == 1)AND (line_fin == 1)) then
   dot_cnt    = 0
elsif (dot_active == 1) then
   dot_cnt    = dot_cnt + 1
else
   dot_cnt    = dot_cnt
// calculate the word select bits
```

```
bit_sel[5:0]                    := dot_cnt[5:0]
```

The dot generator also maintains a read buffer pointer which is incremented each time a 64-bit word is processed. The pointer is used to address the correct 64-bit dot data word within the DIU buffers. The pointer is reset when *llu_go_pulse* is 1. Unlike the dot counter the read pointer is not reset each line but rounded up the nearest 256-bit word. This allows for more efficient use of the DIU buffers at line finish.

When the dot counter reaches the join point for the dot generator (*join_point*), it jumps to the next 256 bit word in the DIU buffer but continues to read from the next bit position within that word. If the join point coincides with a word boundary, no 256-bit increment is required.

```
// read pointer logic
if (llu_go_pulse == 1) then
   read_adr = 0
elsif ((dot_active == 1)AND((dot_cnt[7:0] == 255)OR(line_fin
== 1)))then
   // end of line round up
   read_adr[3:2] ++
   read_adr[1:0] = 0
elsif        ((dot_active        ==        1)AND(dot_cnt        ==
join_point)AND(dot_cnt[5:0] == 63)) then
   // join point jump 256 bits
   read_adr[1:0] ++                                          //
regular increment
   read_adr[3:2] ++                                    // join
point 256 increment
elsif        ((dot_active        ==        1)AND(dot_cnt        ==
join_point)AND(dot_cnt[5:0] != 63)) then
   // join point jump 256 bits, bottom bits remain the same
   read_adr[3:2] ++                                    // join
point 256 increment only
elsif ((dot_active == 1)AND(dot_cnt[5:0] == 63)) then
   .read_adr[3:0] ++                                         //
regular increment
```

31.7.5   Fifo fill level

The LLU keeps a running total of the number of lines in the dot line store FIFO. Every time the DWU signals a line end (*dwu_llu_line_wr* active pulse) it increments the *filllevel*. Conversely if the LLU detects a line end (*line_rd* pulse) the *filllevel* is decremented and the line read is signalled to the DWU via the *llu_dwu_line_rd* signal.

The LLU fill level block is used to determine when the dot line has enough data stored before the LLU should begin to start reading. The LLU at page start is disabled. It waits for the DWU to write lines to the dot line FIFO, and for the fill level to increase. The LLU remains disabled until the fill level has reached the programmed threshold (*fifo_read_thres*). When the threshold is reached it signals the LLU to start processing the page by setting *llu_en* high. Once the LLU has started

processing dot data for a page it will not stop if the *filllevel* falls below the threshold, but will stall is *filllevel* falls to zero.

The line fifo fill level can be read by the CPU via the PCU at any time by accessing the *FifoFillLevel* register. The CPU must toggle the *Go* register in the LLU for the block to be correctly initialized at page start and the fifo level reset to zero.

```
if (llu_go_pulse == 1) then
   filllevel = 0
elsif ((line_rd == 1) AND (dwu_llu_line_wr == 1)) then
   // do nothing
elsif (line_rd == 1) then
   filllevel --
elsif (dwu_llu_line_wr == 1) then
   filllevel ++
// determine the threshold, and set the LLU going
if (llu_go_pulse == 1) OR (filllevel == 0 )) then
   llu_en = 0
elsif (filllevel == fifo_read_threshold ) then
   llu_en = 1
```

31.7.6 · DIU interface

31.7.6.1 DIU interface description

The DIU interface block is responsible for determining when dot data needs to be read from DRAM, keeping the dot generators supplied with data and calculating the DRAM read address based on configured parameters, FIFO fill levels and position in a line.

The fill level block enables DIU requests by activating *llu_en* signal. The DIU interface controller then issues requests to the DIU for the LLU buffers to be filled with dot line data (or fill the LLU buffers with null data without requesting DRAM access, if required).

At page start the DIU interface determines which buffers should be filled with null data and which should request DRAM access. New requests are issued until the dot line is completely read from DRAM.

For each request to the DRAM the address generator calculates where in the DRAM the dot data should be read from. The *color_enable* bus determines which colors are enabled, the interface never issues DRAM requests for disabled colors.

31.7.6.2 Interface controller

The interface controller co-ordinates and issues requests for data transfers from DRAM. The state machine waits in *Idle* state until it is enabled by the LLU controller (*llu_en*) and a request for data transfer is received from the write pointer block.

When an active request is received (*req_active* equals 1) the state machine jumps to the *ColorSelect* state to determine which colors (*color_cnt*) in the group need a data transfer. A group is defined as all odd colors or all even colors. If the color isn't enabled (*color_enable*) the count just increments, and no data is transferred. If the color is enabled, the state machine takes one of

two options, either a null data transfer or an actual data transfer from DRAM. A null data transfer writes zero data to the DIU buffer and does not issue a request to DRAM.

The state machine determines if a null transfer is required by checking the *color_start* signal for that color.

If a null transfer is required the state machine doesn't need to issue a request to the DIU and so jumps directly to the data transfer states (*Data0* to *Data3*). The machine clocks through the 4 states each time writing a null 64-bit data word to the buffer. Once complete the state machine returns to the *ColorSelect* state to determine if further transfers are required.

If the *color_start* is active then a data transfer is required. The state machine jumps to the *Request* state and issue a request to the DIU controller for DRAM access by setting *llu_diu_rreq* high. The DIU responds by acknowledging the request (*diu_llu_rack* equals 1) and then sending 4 64-bit words of data. The transition from *Request* to *Data0* state signals the address generator to update the address pointer (*adr_update*). The state machine clocks through *Data0* to *Data3* states each time writing the 64-bit data into the buffer selected by the *req_sel* bus. Once complete the state machine returns to the *ColorSelect* state to determine if further transfers are required.

When in the *ColorSelect* state and all data transfers for colors in that group have been serviced (i.e. when *color_cnt* is 6) the state machine will return to the *Idle* state. On transition it will update the word counter logic (*word_dec*) and enabled the request logic (*req_update*).

A reset or *llu_go_pulse* set to 1 will cause the state machine to jump directly to *Idle*. The controller will remain in *Idle* state until it is enabled by the LLU controller via the *llu_en* signal. This prevents the DIU attempting the fill the DIU buffers before the dot line store FIFO has filled over its threshold level.

31.7.6.3 Color activate

The color activate logic maintains an absolute line count indicating the line number currently being processed by the LLU. The counter is reset when the *llu_go_pulse* is 1 and incremented each time a *line_rd* pulse is received. The count value (*line_cnt*) is used to determine when to start reading data for a color.

The count is implemented as follows:

```
if ( llu_go_pulse == 1) then
    line_cnt = 0
elsif ( line_rd == 1) then
    line_cnt ++
```

The color activate logic compares line count with the relative line value to determine when the LLU should start reading data from DRAM for a particular half color. It signals the interface controller block which colors are active for this dot line in a page (via the *color_start* bus). It is used by the interface controller to determine which DIU buffers require null data.

Once the *color_start* bit for a color is set it cannot be cleared in the normal page processing process. The bits must be reset by the CPU at the end of a page by transitioning the *Go* bit and causing a pulse on the *llu_go_pulse* signal.

Any color not enabled by the *color_enable* bus will never have its *color_start* bit set.

```
for (i=0; i<12;i++) {
```

511

```
                    if ( llu_go_pulse == 1) then
                       col_on[i] = 0
                    elsif ( color_enable[i % 6] == 1 ) then
                       col_on[i] = 0
5                   elsif ( line_cnt == color_rel_line[i]) then
                       col_on[i] = 1
                    }
                 // select either odd or even colors
                 if ( odd_even_sel == 1 ) then  // odd selected
10                  color_start[5:0]                                      =
                 {col_on[11],col_on[9],col_on[7],col_on[5],col_on[3],col_on[1
                 ]}
                 else                             // even selected
                    color_start[5:0]                                     =
15               {col_on[10],col_on[8],col_on[6],col_on[4],col_on[2],col_on[0
                 ]}
```

31.7.6.4 Address generator

The address generator block maintains 24 pointers (*current_adr_a[11:0]* and *current_adr_b[11:0]*) to DRAM corresponding to 2 read addresses in the dot line FIFO for each half color. The *current_adr_a* group of pointers are used when the dot generator is feeding printhead channel A, and the *current_adr_b* group of pointers are used when the dot generator is feeding printhead channel B. For each DRAM access the 2 address pointers are updated but only one can be used for an access. The word counter block determines which pointer group should be used to access DRAM, via the pointer select signals (*ptr_sel*). In certain cases (e.g. the join point is not 256-bit aligned and the word is on the join point) the address pointers should not be updated for an access, the word counter block determines the exception cases and indicates to the address generator to skip the update via the *join_stall* signal.

When a DRAM transfer occurs the address pointer is used first and then updated for the next transfer for the color. The pointer used is selected by the *req_sel* and *ptr_sel* buses, and the pointer update is initiated by the *adr_update* signal from the interface controller.

The address update is calculated as follows (pointer group A logic is shown but the same logic is used to update the B pointer group a clock cycle later):

```
                 // update the A pointers
35               if (ptra_wr_en == 1) then                // write from the
                 configuration block
                    current_adr_a[ptr_adr] = ptr_wr_data;
                 elsif ( adr_update_a == 1) then {   // address update from
                 state machine
40                  if ((req_sel == NULL )OR (join_stall == 1)) then
                       // do nothing
                    else
                       // temporary variable setup
                       next_adr  = current_adr_a[req_sel] + 1
```

```
                start_adr = color_base_adr[req_sel]
                end_adr   = color_base_adr[req_sel + 1]
                // determine how to update the pointer
                if (next_adr == end_adr) then
                    current_adr_a[req_sel] = start_adr
                else
                    current_adr_a[req_sel] = next_adr
        }
```

The correct address to use for a transfer is selected by the *ptr_sel* signals from the word counter block. They indicate which set of address pointers should be used based on the current word being transferred from the DRAM and the configured join point values (*join_word*).

```
                // select the address pointer to use for access
                if (req_sel[0] == 1) then                        // odd
                pointer selector
                    if (ptr_sel[1] == 1) then
                        llu_diu_radr = current_adr_b[req_sel]     // latter part
                of line
                        else
                        llu_diu_radr = current_adr_a[req_sel]     // former part
                of line
                else                                              // even
                pointer selector
                    if (ptr_sel[0] == 1) then
                        llu_diu_radr = current_adr_b[req_sel]     // latter part
                of line
                        else
                        llu_diu_radr = current_adr_a[req_sel]     // former part
                of line
```

31.7.6.5 Write pointer

The write pointer logic maintains the buffer write address pointers, determines when the DIU buffers need a data transfer and signals when the DIU buffers are empty. The write pointer determines the address in the DIU buffer that the data should be transferred to.

The write pointer logic compares the read and write pointers of each DIU buffer to determine which buffers require data to be transferred from DRAM, and which buffers are empty (the *buf_emp* signals).

Buffers are grouped into odd and even buffers, if an odd buffer requires DRAM access the *odd_pend* signals will be active, if an even buffer requires DRAM access the *even_pend* signals will be active. If both odd and even buffers require DRAM access at exactly the same time, the even buffers will get serviced first. If a group of odd buffers are being serviced and an even buffer becomes pending, the odd group of buffers will be completed before the starting the even group, and vice versa.

If any buffer requires a DRAM transfer, the logic will indicate to the interface controller via the *req_active* signal, with the *odd_even_sel* signal determining which group of buffers get serviced. The interface controller will check the *color_enable* signal and issue DRAM transfers for all enabled colors in a group. When the transfers are complete it tells the write pointer logic to update

5 the request pending via *req_update* signal.

The *req_sel[3:0]* signal tells the address generator which buffer is being serviced, it is constructed from the *odd_even_sel* signal and the *color_cnt[2:0]* bus from the interface controller. When data is being transferred to DRAM the word pointer and write pointer for the corresponding buffer are updated. The *req_sel* determines which pointer should be incremented.

10 The write pointer logic operates the same way regardless of whether the transfer is null or not.

```
// determine which buffers need updates
buf_emp[1:0] = 0
odd_pend    = 0
even_pend   = 0
if ( wr_adr[0][3:2] == rd_adr[0][3:2] )
    even_pend = 1
if ( wr_adr[1][3:2] == rd_adr[1][3:2] )
    odd_pend  = 1
// determine if buffers are empty
if ((wr_adr[0][3:0] == rd_adr[0][3:0])) then
    buf_emp[0] = 1
if ((wr_adr[1][3:0] == rd_adr[1][3:0])) then
    buf_emp[1] = 1
// fixed  servicing  order,  only  update  when  controller
dictates so
if (req_update == 1) then {
   if (even_pend == 1) then       // even always first
      odd_even_sel = 0
      req_active   = 1
   elsif (odd_pend == 1 ) then    // then check odd
      odd_even_sel = 0
      req_active   = 1
   else                           // nothing active
      odd_even_sel = 0
      req_active   = 0
   }
// selected requestor
req_sel[3:0]    =    {color_cnt[2:0],odd_even_sel}          //
concatentation
```

The write address pointer logic consists of 2 2-bit counters and a word select pointer. The counters are reset when *llu_go_pulse* is one. The word pointer (*word_ptr*) is common to all buffers and is used to write   64-bit words into the DIU buffer. It is incremented when *buf_rd_en* is active.

514

When a group of buffers are updated the state machine increments the write pointer (*wr_ptr[odd_even_sel]*) via the *group_fin* signal. A concatenation of the write pointer and the word pointer are use to construct the buffer write address. The write pointers are not reset at the end of each line.

5

```
// determine which pointer to update
if (llu_go_pulse == 1) then
   wr_ptr[1:0]  = 0
   word_ptr     = 0
elsif (buf_rd_en == 1) then
   word_ptr++
   wr_en[req_sel] = 1
elsif (group_fin = 1 ) then
   wr_ptr[odd_even_sel]++
```

10

15

```
// create the address from the write pointer and word
pointer
wr_adr[odd_even_sel] = {wr_ptr[odd_even_sel],word_ptr}    //
concatenation
```

20

31.7.6.6 Word count

The word count logic maintains 2 counters to track the number of words transferred from DRAM per line, one counter for odd data, and one counter for even. On receipt of a *llu_go_pulse*, the counters are initialized to a *join_word* value (number of words to the join point for that printhead channel) and the pointer select values to zero (*ptr_sel*). When a group of words are transferred to DRAM as indicated by the *word_dec* signal from the interface controller, the corresponding counter is decremented. The counter to decrement is indicated by the *odd_even_sel* signal from the write pointer block (even = 0, odd = 1).

When a counter is zero and the *ptr_sel* is zero, the counter is re-initialized to the second *join_word* value and *ptr_sel* is inverted. The counter continues to count down to zero each time a *word_dec* signal is received. When a counter is zero and the *ptr_sel* is one, it signals the end of a line (the *last_wd* signal) and initializes the counter to the first *join_point* value for the next line transfer.

The *ptr_sel* signal is used in the address generator to select the correct address pointer to use for that particular access.

35

```
// determine which counter to decrement
if (llu_go_pulse == 1) then
   word_cnt[0]  = join_word[0]          // even count
   ptr_sel[0]   = 0                            // even
generator starts with pointer A
   word_cnt[1]  = join_word[1]          // odd count
   ptr_sel[1]   = 0                     // odd generator
starts with pointer A
```

40

```
            elsif (word_dec == 1) then {                    // need to
            decrement one word counter
               if (odd_even_sel == 0) then                  // even counter
            update
                  if (word_cnt[0] == 0) then
                     word_cnt[0] = join_word[ptr_sel[0]]   // re-initialize
            pointer
                     ptr_sel[0]  = ~(ptr_sel[0])
                     if (ptr_sel[0]== 1) then               // determine if
            this the last word
                        last_wd  = 1
                  else
                     word_cnt[0] --                          // normal
            decrement
               else                                         // odd counter
            update
                  if (word_cnt[1] == 0) then
                     word_cnt[1] = join_word[ptr_sel[1]]   // re-initialize
            pointer
                     ptr_sel[1]  = ~(ptr_sel[1])
                     if (ptr_sel[1]== 1) then               // determine if
            this the last word
                        last_wd  = 1
                  else
                     word_cnt[1] --                          // normal
            decrement
                  }
```

The word count logic also determines if the current word to be transferred is the join word, and if so it determines if it is aligned on a 256-bit boundary or not. If the join point is aligned to a boundary there is no need to prevent the address counter from incrementing, otherwise the address pointers are stalled for that word transfer (*join_stall*).

```
            join_stall = (((ptr_sel[0]  == 0)AND  (word_cnt[0]  == 0)AND
            (join_point[0][7:0] != 0))
                        AND  ((ptr_sel[1]  == 0)AND  (word_cnt[1]  == 0)AND
            (join_point[1][7:0] != 0)))
```

The word count logic also determines when a complete line has been read from DRAM, it then signals the fifo fill level logic in both the LLU and DWU (via *line_rd* signal) that a complete line has been read by the LLU (*llu_dwu_line_rd*).

```
                 // line finish logic
             if (llu_go_pulse == 1) then
                line_fin = 0
                line_rd  = 0
             elsif ((last_wd == 1) AND (line_fin == 0)) then
```

```
          line_fin = 1                          // first group last_wd
       finish pulse
          line_rd  = 0
       elsif ((last_wd == 1) AND (line_fin == 1)) then
          line_fin = 0                           // second group last_wd
       finish pulse
          line_rd  = 1
       else
          line_fin = line_fin              // stay the same
          line_rd  = 0
```

32   PrintHead Interface (PHI)

32.1   OVERVIEW

The Printhead interface (PHI) accepts dot data from the LLU and transmits the dot data to the printhead, using the printhead interface mechanism. The PHI generates the control and timing signals necessary to load and drive the bi-lithic printhead. The CPU determines the line update rate to the printhead and adjusts the line sync frequency to produce the maximum print speed to account for the printhead IC's size ratio and inherent latencies in the syncing system across multiple SoPECs.

The PHI also needs to consider the order in which dot data is loaded in the printhead. This is dependent on the construction of the printhead and the relative sizes of printhead ICs used to create the printhead. See Bi-lithic Printhead Reference document for a complete description of printhead types [10].

The printing process is a real-time process. Once the printing process has started, the next printline's data must be transferred to the printhead before the next line sync pulse is received by the printhead. Otherwise the printing process will terminate with a buffer underrun error.

The PHI can be configured to drive a single printhead IC with or without synchronization to other SoPECs. For example the PHI could drive a single IC printhead (i.e. a printhead constucted with one IC only), or dual IC printhead with one SoPEC device driving each printhead IC.

The PHI interface provides a mechanism for the CPU to directly control the PHI interface pins, allowing the CPU to access the bi-lithic printhead to:

- determine printhead temperature
- test for and determine dead nozzles for each printhead IC
- initialize each printhead IC
- pre-heat each printhead IC

Figure 277 shows a high level data flow diagram of the PHI in context.

32.2   PRINTHEAD MODES OF OPERATION

The printhead has 8 different modes of operations (although some modes are re-used). The mode of operation is defined by the state of the output pins *phi_lsyncl* and *phi_readl* and the internal printhead mode register. The modes of operation are defined in Table 210.

Table 210. Printhead modes of operation

517

Name	Internal Mode	phi_readl	phi_lsyncl	State	Description
NORMAL	XXX	1	1	N/A	Normal print mode, dot data is clocked into the printhead shift register, on each falling edge of *phi_srclk*
DOT_LOAD/ FIRE_INIT	XXX	1	0	*phi_frclk=0*	Dot Load Mode, data stored in the dot shift register is transferred into the dot latch on the falling edge of *phi_lsyncl,* and latched in on the rising edge of *phi_lsyncl*
				phi_srclk=1	Fire load mode. Parameter for generating fire pattern are loaded into generator, data on *phi_ph_data[1:0][0]* is clocked into the generator on each rising edge of *phi_frclk*
NOZZLE_RESET	001	0	1	N/A	Reset Nozzle Test mode. Reset the state on nozzle test.
CMOS_TEST	111	0	1	N/A	CMOS test mode.
FIRE_GEN	000	0	1	N/A	Fire Initialise mode. The initialised generator creates the fire pattern and shift select pattern. The pattern is clocked into the fire shift register and select shift register on the rising edge of *phi_frclk*
TEMP_TEST	010	0	0	N/A	Temperature test output.
NOZZLE_TEST	001	0	0	N/A	Nozzle test output. The result of a nozzle test is output on *phi_frclk_i.*

32.3   DATA RATE EQUALIZATION

The LLU can generate dot data at the rate of 12 bits per cycle, where a cycle is at the system clock frequency. In order to achieve the target print rate of 30 sheets per minute, the printhead needs to print a line every 100μs (calculated from 300mm @ 65.2 dots/mm divided by 2 seconds =~ 100μsec). For a 7:3 constructed printhead this means that 9744 cycles at 320Mhz is quick enough to transfer the 6-bit dot data (at 2 bits per cycle). The input FIFOs are used to de-couple the read and write clock domains as well as provide for differences between consume and fill rates of the PHI and LLU.

Nominally the system clock (*pclk*) is run at 160Mhz and the printhead interface clock (*doclk*) is at 320Mhz.

If the PHI was to transfer data at the full printhead interface rate, the transfer of data to the shorter printhead IC would be completed sooner than the longer printhead IC. While in itself this isn't an
5   issue it requires that the LLU be able to supply data at the maximum rate for short duration, this requires uneven bursty access to DRAM which is undesirable. To smooth the LLU DRAM access requirements over time the PHI transfers dot data to the printhead at a pre-programmed rate, proportional to the ratio of the shorter to longer printhead ICs.

The printhead data rate equalization is controlled by *PrintHeadRate[1:0]* registers (one per
10   printhead IC). The register is a 16 bit bitmap of active clock cycles in a 16 clock cycle window. For example if the register is set to 0xFFFF then the output rate to the printhead will be full rate, if it's set to 0xF0F0 then the output rate is 50% where there is 4 active cycles followed by 4 inactive cycles and so on. If the register was set to 0x0000 the rate would be 0%. The relative data transfer rate of the printhead can be varied from 0-100% with a granularity of 1/16 steps.

15   Table 211. Example rate equalization values for common printheads

Printhead Ratio A:B	Printhead A rate (%)	Printhead B rate (%)
8:2	0xFFFF (100%)	0x1111 (25%)
7:3	0xFFFF (100%)	0x5551 (43.7%)
6:4	0xFFFF (100%)	0xF1F2 (68.7%)
5:5	0xFFFF (100%)	0xFFFF (100%)

If both printhead ICs are the same size (e.g. a 5:5 printhead) it may be desirable to reduce the data rate to both printhead ICs, to reduce the read bandwidth from the DRAM.
20   32.4   DOT GENERATE AND TRANSMIT ORDER
Several printhead types and arrangements exists (see [10] for other arrangements). The PHI is capable of driving all possible configurations, but for the purposes of simplicity only one arrangement (arrangement 1 - see [10] for definition) is described in the following examples.
The structure of the printhead ICs dictate the dot transmit order to each printhead IC. The PHI
25   accepts two streams of dot data from the LLU, one even stream the other odd. The PHI constructs the dot transmit order streams from the dot generate order received from the LLU. Each stream of data has already been arranged in increasing or decreasing dot order sense by the DWU. The exact sense choice is dependent on the type of printhead ICs used to construct the printhead, but regardless of configuration the odd and even stream should be of opposing sense.
30   The dot transmit order is shown in Figure 281. Dot data is shifted into the printhead in the direction of the arrow, so from the diagram (taking the type 0 printhead IC) even dot data is transferred in increasing order to the mid point first (0, 2, 4, ..., m-6, m-4, m-2), then odd dot data in decreasing order is transferred (m-1, m-3, m-5,...., 5, 3, 1). For the type 1 printhead IC the order is reversed, with odd dots in increasing order transmitted first, followed by even dot data in

decreasing order. Note for any given color the odd and even dot data transferred to the printhead ICs are from different dot lines, in the example in the diagram they are separated by 5 dot lines. Table 212 shows the transmit dot order for some common A4 printheads. Different type printheads may have the sense reversed and may have an odd before even transmit order or vice versa.

Table 212. Example printhead ICs, and dot data transmit order for A4 (13824 dots) page

Size	Dots	Dot Order	
Type 0 Printhead IC			
8	11160	0,2,4,8......,5574,5576,5578	5579,5577,5575......7,5,3,1
7	9744	0,2,4,8......,4866,4868,4870	4871,4869,4867.....7,5,3,1
6	8328	0,2,4,8......,4158,4160,4162	4163,4161,4159......7,5,3,1
5	6912	0,2,4,8......,3450,3452,3454	3455,3453,3451......7,5,3,1
4	5496	0,2,4,8......,2742,2744,2746	2847,2845,2843......7,5,3,1
3	4080	0,2,4,8......,2034,2036,2038	2039,2037,2035......7,5,3,1
2	2664	0,2,4,8......,1326,1328,1330	1331,1329,1327......7,5,3,1
Type 1 Printhead IC			
8	11160	13823,13821,13819 ......,1337,1335,1333	1332,1334,1336.......13818,138 20,13822
7	9744	13823,13821,13819 ......,2045,2043,2041	2040,2042,2044.......13818,138 20,13822
6	8328	13823,13821,13819 ......,2853,2851,2849	2848,2850,2852.......13818,138 20,13822
5	6912	13823,13821,13819 ......,3461,3459,3457	3456,3458,3460.......13818,138 20,13822
4	5496	13823,13821,13819 ......,4169,4167,4165	4164,4166,4168.......13818,138 20,13822
3	4080	13823,13821,13819 ......,4877,4875,4873	4872,4874,4876.......13818,138 20,13822
2	2664	13823,13821,13819 ......,5585,5583,5581	5580,5582,5584.......13818,138 20,13822

32.4.1   Dual Printhead IC

The LLU contains 2 dot generator units. Each dot generator reads dot data from DRAM and generates a stream of dots in increasing or decreasing order. A dot generator can be configured to produce odd or even dot data streams, and the dot sense is also configurable. In Figure 281 the odd dot generator is configured to produce odd dot data in decreasing order and the even dot generator produces dot data in increasing order. The LLU takes care of any vertical misalignment between the 2 printhead ICs, presenting the PHI with the appropriate data ready to be transmitted to the printhead.

In order to reconstruct the dot data streams from the generate order to the transmit order, the connection between the generators and transmitters needs to be switched at the mid point. At line start the odd dot generator feeds the type 1 printhead, and the even dot generator feeds the type 0 printhead. This continues until both printheads have received half the number of dots they require (defined as the mid point). The mid point is calculated from the configured printhead size registers (*PrintHeadSize*). Once both printheads have reached the mid point, the PHI switches the connections between the dot generators and the printhead, so now the odd dot generator feeds the type 0 printhead and the even dot generator feeds the type 1 printhead. This continues until the end of the line.

It is possible that both printheads will not be the same size and as a result one dot generator may reach the mid point before the other. In such cases the quicker dot generator is stalled until both dot generators reach the mid point, the connections are switched and both dot generators are restarted.

Note that in the example shown in Figure 281 the dot generators could generate an A4 line of data in 6912 cycles, but because of the mismatch in the printhead IC sizes the transmit time takes 9744 cycles.

32.4.2   Single printhead IC

In some cases only one printhead IC may be connected to the PHI. In Figure 282 the dot generate and transmit order is shown for a single IC printhead of 9744 dots width. While the example shows the printhead IC connected to channel A, either channel could be used. The LLU generates odd and even dot streams as normal, it has no knowledge of the physical printhead configuration. The PHI is configured with the printhead size (*PrintHeadSize[1]* register) for channel B set to zero and channel A is set to 9744.

Note that in the example shown in Figure 283 the dot generators could generate an 7 inch line of data in 4872 cycles, but because the printhead is using one IC, the transmit time takes 9744 cycles, the same speed as an A4 line with a 7:3 printhead.

32.4.3   Summary of generate and transmit order requirements

In order to support all the possible printhead arrangements, the PHI (in conjuction with the LLU/DWU) must be capable of re-ordering the bits according to the following criteria:

- Be able to output the even or odd plane first.
- Be able to output even and odd planes independently.
- Be able to reverse the sequence in which the color planes of a single dot are output to the printhead.

32.5   PRINT SEQUENCE

The PHI is responsible for accepting dot data streams from the LLU, restructuring the dot data sequence and transferring the dot data to each printhead within a line time (i.e before the next line sync).

Before a page can be printed the printhead ICs must be initialized. The exact initialization sequence is configuration dependent, but will involve the fire pattern generation initialization and other optional steps. The initialization sequence is implemented in software.

Once the first line of data has been transferred to the printhead, the PHI will interrupt the CPU by asserting the *phi_icu_print_rdy* signal. The interrupt can be optionally masked in the ICU and the CPU can poll the signal via the PCU or the ICU. The CPU must wait for a print ready signal in all printing SoPECs before starting printing.

5   Once the CPU in the PrintMaster SoPEC is satisfied that printing should start, it triggers the LineSyncMaster SoPEC by writing to the *PrintStart* register of all printing SoPECs. The transition of the *PrintStart* register in the LineSyncMaster SoPEC will trigger the start of *lsyncl* pulse generation. The PrintMaster and LineSyncMaster SoPEC are not necessarily the same device, but often are the same. For a more in depth definition see section 12.1.1 Multi-SoPEC systems on

10 page 105.

Writing a 1 to the *PrintStart* register enables the generation of the line sync in the LineSyncMaster which is in turn used to align all SoPECs in a multi-SoPEC system. All printhead signaling is aligned to the line sync. The *PrintStart* is only used to align the first line sync in a page.

When a SoPEC receives a line sync pulse it means that the line previously transferred to the

15 printhead is now printing, so the PHI can begin to transfer the next line of data to the printhead. When the transfer is complete the PHI will wait for the next line sync pulse before repeating the cycle. If a line sync arrives before a complete line is transferred to the printhead (i.e. a buffer error) the PHI generates a buffer underrun interrupt, and halts the block.

For each line in a page the PHI must transfer a full line of data to the printhead before the next

20 line sync is generated or received.

32.5.1   Sync pulse control

If the PHI is configured as the LineSyncMaster SoPEC it will start generating line sync signals *LsyncPre* number of *pclk* cycles after *PrintStart* register rising transition is detected. All other signals in the PHI interface are referenced from the rising edge of *phi_lsyncl* signal.

25 If the SoPEC is in line sync slave mode it will receive a line sync pulse from the LineSyncMaster SoPEC through the *phi_lsyncl* pin which will be programmed into input mode. The *phi_lsyncl* input pin is treated as an asynchronous input and is passed through a de-glitch circuit of programmable de-glitch duration (*LsyncDeglitchCnt*).

The *phi_lsyncl* will remain low for *LsyncLow* cycles, and then high for *LsyncHigh* cycles. The

30 *phi_lsyncl* profile is repeated until the page is complete. The period of the *phi_lsyncl* is given by *LsyncLow* + *LsyncHigh* cycles. Note that the *LsyncPre* value is only used to vary the time between the generation of the first *phi_lsyncl* and the *PageStart* indication from the CPU. See Figure 284 for reference diagram.

If the SoPEC device is in line sync slave mode, the *LsyncHigh* register specifies the minimum

35 allowed *phi_lsyncl* period. Any *phi_lsyncl* pulses received before the *LsyncHigh* has expired will trigger a buffer underrun error.

32.5.2   Shift register signal control

Once the PHI receives the line sync pulse, the sequence of data transfer to the printhead begins. All PHI control signals are specified from the rising edge of the line sync.

The *phi_srclk* (and consequently *phi_ph_data*) is controlled by the *SrclkPre*, *SrclkPost* registers. The *SrclkPre* specifies the number of *pclk* cycles to wait before beginning to transfer data to the printhead. Once data transfer has started, the profile of the *phi_srclk* is controlled by *PrintHeadRate* register and the status of the PHI input FIFO. For example it is possible that the input FIFO could empty and no data would be transferred to the printhead while the PHI was waiting. After all the data for a printhead is transferred to the PHI, it counts *SrclkPost* number of *pclk* cycles. If a new *phi_lsync* falling edge arrives before the count is complete the PHI will generate a buffer underrun interrupt (*phi_icu_underrun*).

32.5.3 Firing sequence signal control

The profile of the *phi_frclk* pulses per line is determined by 4 registers *FrclkPre*, *FrclkLow*, *FrclkHigh*, *FrclkNum*. The *FrclkPre* register specifies the number of cycles between line sync rising edge and the *phi_frclk* pulse high. It remains high for *FrclkHigh* cycles and then low for *FrclkLow* cycles. The number of pulses generated per line is determined by *FrclkNum* register. The total number of cycles required to complete a firing sequence should be less than the *phi_lsync* period i.e. $((FrclkHigh + FrclkLow) * FrclkNum) + FrclkPre < (LsyncLow + LsyncHigh)$. Note that when in CPU direct control mode (*PrintHeadCpuCtrl*=1) and *PrintHeadCpuCtrlMode[x]* =1, the *frclk* generator is triggered by the transition of the *FireGenSoftTrigger[0]* bit from 0 to 1. Figure 284 details the timing parameters controlling the PHI. All timing parameters are measured in number of *pclk* cycles.

32.5.4 Page complete

The PHI counts the number of lines processed through the interface. The line count is initialised to the *PageLenLine* and decrements each time a line is processed. When the line count is zero it pulses the *phi_icu_page_finish* signal. A pulse on the *phi_icu_page_finish* automatically resets the PHI *Go* register, and can optionally cause an interrupt to the CPU. Should the page terminate abnormally, i.e. a buffer underrun, the *Go* register will be reset and an interrupt generated.

32.5.5 Line sync interrupt

The PHI will generate an interrupt to the CPU after a predefined number of line syncs have occured. The number of line syncs to count is configured by the *LineSyncInterrupt* register. The interrupt can be disabled by setting the register to zero.

32.6 DOT LINE MARGIN

The PHI block allows the generation of margins either side of the received page from the LLU block. This allows the page width used within PEP blocks to differ from the physical printhead size.

This allows SoPEC to store data for a page minus the margins, resulting in less storage requirements in the shared DRAM and reduced memory bandwidth requirements. The difference between the dot data line size and the line length generated by the PHI is the dot line margin length. There are two margins specified for any sheet, a margin per printhead IC side. The margin value is set by programming the *DotMargin* register per printhead IC. It should be noted that the *DotMargin* register represents half the width of the actual margin (either left or right margin depending on paper flow direction). For example, if the margin in dots is 1 inch (1600

dots), then *DotMargin* should be set to 800. The reason for this is that the PHI only supports margin creation cases 1 and 3 described below.

See example in Figure 284.

In the example the margin for the type 0 printhead IC is set at 100 dots (*DotMargin*==100),

5   implying an actual margin of 200 dots.

If case one is used the PHI takes a total of 9744 *phi_srclk* cycles to load the dot data into the type 0 printhead. It also requires 9744 dots of data from the LLU which in turn gets read from the DRAM. In this case the first 100 and last 100 dots would be zero but are processed though the SoPEC system consuming memory and DRAM bandwidth at each step.

10 In case 2 the LLU no longer generates the margin dots, the PHI generates the zeroed out dots for the margining. The *phi_srclk* still needs to toggle 9744 times per line, although the LLU only needs to generate 9544 dots giving the reduction in DRAM storage and associated bandwidth. The case 2 senario is not supported by the PHI because the same effect can be supported by means of case 1 and case 3.

15 If case 3 is used the benefits of case 2 are achieved, but the *phi_srclk* no longer needs to toggle the full 9744 clock cycles. The *phi_srclk* cycles count can be reduced by the margin amount (in this case 9744-100=9644 dots), and due to the reduction in *phi_srclk* cycles the *phi_lsyncl* period could also be reduced, increasing the line processing rate and consequently increasing print speed. Case 3 works by shifting the odd (or even) dots of a margin from line Y to become the

20 even (or odd) dots of the margin for line Y-4, (Y-5 adjusted due to being printed one line later). This works for all lines with the exception of the first line where there has been no previous line to generate the zeroed out margin. This situation is handled by adding the line reset sequence to the printhead initialization procedure, and is repeated between pages of a document.

32.7   DOT COUNTER

25 For each color the PHI keeps a dot usage count for each of the color planes (called *AccumDotCount*). If a dot is used in particular color plane the corresponding counter is incremented. Each counter is 32 bits wide and saturates if not reset. A write to the *DotCountSnap* register causes the *AccumDotCount[N]* values to be transferred to the *DotCount[N]* registers (where N is 5 to 0, one per color). The *AccumDotCount* registers are cleared on value transfer.

30 The *DotCount[N]* registers can be written to or read from by the CPU at any time. On reset the counters are reset to zero.

The dot counter only counts dots that are passed from the LLU through the PHI to the printhead. Any dots generated by direct CPU control of the PHI pins will not be counted.

32.8   CPU IO CONTROL

35 The PHI interface provides a mechanism for the CPU to directly control the PHI interface pins, allowing the CPU to access the bi-lithic printhead:

- Determine printhead temperature
- Test for and determine dead nozzles for each printhead IC
- Printhead IC initialization

40 - Printhead pre-heat function

The CPU can gain direct control of the printhead interface connections by setting the *PrintHeadCpuCtrl* register to one. Once enabled the printhead bits are driven directly by the *PrintHeadCpuOut* control register, where the values in the register are reflected directly on the printhead pins and the status of the printhead input pins can be read directly from the *PrintHeadCpuIn*. The direction of pins is controlled by programming *PrintHeadCpuDir* register. The register to pin mapping is as follows:

Table 213. CPU control and status registers mapping to printhead interface

Register Name	bits	Printhead pin
PrintHeadCpuOut	0	phi_lsyncl_o
	1	phi_frclk_o
	2	Reserved
	4:3	phi_ph_data_o[0][1:0]
	6:5	phi_ph_data_o[1][1:0]
	8:7	phi_srclk[1:0]
	9	phi_readl
PrintHeadCpuDir	0	phi_lsyncl_e direction control 1 - output mode 0 - input mode
	1	phi_frclk_e direction control 1 - output mode 0 - input mode
	2	Reserved
PrintHeadCpuIn	0	phi_lsyncl_i
	1	phi_frclk_i
	2	Reserved

It is important to note that once in *PrintHeadCpuCtrl* mode it is the responsibility of the CPU to drive the printhead correctly and not create situations where the printhead could be destroyed such as activating all nozzles together.

The *phi_srclk* is a double data rate clock (DDR) and as such will clock data on both edges in the printhead.

Note the following procedures are based on current printhead capabilities, and are subject to change.

32.9   IMPLEMENTATION

32.9.1   Definitions of I/O

Table 214. Printhead interface I/O definition

Port name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	System Clock

Doclk	1	In	Data out clock (2x *pclk*) used to transfer data to printhead
prst_n	1	In	System reset, synchronous active low. Synchronous to *pclk*
dorst_n	1	In	System reset, synchronous active low. Synchronous to *doclk*
General			
phi_icu_print_rdy	1	Out	Indicates that the first line of data is transferred to the printhead Active high.
phi_icu_page_finish	1	Out	Indicates that data for a complete page has transferred. Active high
phi_icu_underrun	1	Out	Indicates the PHI has detected a buffer underrun. Active high
phi_icu_linesync_int	1	Out	Indicates the PHI has detected *LineSyncInterrupt* number of line syncs.
Debug			
debug_data_valid	1	In	Output debug data valid to be muxed on to the PHI pin
debug_cntrl	1	In	Control signal for the PHI to indicate whether or not the debug data valid (and *pclk*) should be selected by the pin mux. Active high.
LLU Interface			
llu_phi_data[1:0][5:0]	2x6	In	Dot Data from LLU to the PHI, each bit is a color plane 5 downto 0. Bus 0 - Even dot data stream Bus 1 - Odd dot data stream Data is active when corresponding bit is active in *llu_phi_avail* bus
phi_llu_ready[1:0]	2	Out	Indicates that PHI is ready to accept data from the LLU 0 - Even dot data stream 1 - Odd dot data stream
llu_phi_avail[1:0]	2	In	Indicates valid data present on corresponding *llu_phi_data*. 0 - Even dot data stream 1 - Odd dot data stream
Printhead Interface			
phi_ph_data[1:0][1:0]	2x2	Out	Dot data output to printhead. Each bus to each printhead contains 2 bits of data Bus 0 - Printhead channel A Bus 1 - Printhead channel B

phi_srclk[1:0]	2	Out	Dot data shift clock used to clock in printhead data, data is shifted on both edges of clock(i.e. double data rate DDR). Bus 0 - Printhead channel A Bus 1 - Printhead channel B
phi_readl	1	Out	Common printhead mode control. Used in conjunction with *phi_lsyncl* to determine the printhead mode 0 - SoPEC receiving, printhead driving 1 - SoPEC driving, printhead receiving
phi_frclk_o	1	Out	Common Fire pattern clock needs to toggle once per fire cycle
phi_frclk_e	1	In	*phi_frclk_o* output enable, when high *phi_frclk_o* pin is driving
phi_frclk_l	1	In	*phi_frclk_i* input from printhead
phi_lsyncl_o	1	Out	Capture dot data for next print line, output mode
phi_lsyncl_e	1	In	*phi_lsyncl* output enable, when high *phi_lsyncl* pin is driving
phi_lsyncl_i	1	In	Line Sync Pulse from Master SoPEC
PCU Interface			
pcu_phi_sel	1	In	Block select from the PCU. When *pcu_phi_sel* is high both *pcu_adr* and *pcu_dataout* are valid.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_adr[7:2]	6	In	PCU address bus. Only 6 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
phi_pcu_rdy	1	Out	Ready signal to the PCU. When *phi_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *phi_pcu_datain* is valid.
phi_pcu_datain[31:0]	32	Out	Read data bus to the PCU.

32.9.2   PHI sub-block partition

32.9.3   Configuration registers

The configuration registers in the PHI are programmed via the PCU interface. Refer to section 21.8.2 on page 321 for a description of the protocol and timing diagrams for reading and writing registers in the PHI. Note that since addresses in SoPEC are byte aligned and the PCU only supports 32-bit register reads and writes, the lower 2 bits of the PCU address bus are not required to decode the address space for the PHI. When reading a register that is less than 32 bits wide

zeros should be returned on the upper unused bit(s) of *phi_pcu_datain*. Table 215 lists the configuration registers in the PHI

Table 215. PHI registers description

Address PHI_base+	Register	#bits	Reset	Description
Control Registers				
0x00	Reset	1	0x1	Active low synchronous reset, self de-activating. A write to this register will cause a PHI block reset.
0x04	Go	1	0x0	Active high bit indicating the PHI is programmed and ready to use. A low to high transition will cause PHI block internal state to reset. Will be automatically reset if a page finish or a buffer underrun is detected.
General Control				
0x08	PageLenLine	32	0x0000_0000	Specifies the number of dot lines in a page. Indicates the number of lines left to process in this page while the PHI is running (Working register)
0x0c	PrintStart	1	0x0	A high level enables printing to start via the generation of line syncs in a master, and acceptance of line syncs in a slave. Can be set in advance of the print ready signal.
0x10-0x14	DotMargin[1:0]	2x16	0x0000	Specifies for each printhead IC, the width of the margin in dots divided by 2. Value must be divisible by 2 (i.e. the low bit must be 0) 0 - Printhead IC Channel A 1 - Printhead IC Channel B
0x18-0x2C	DotCount[5:0]	6x32	0x0000_0000	Indicates the number of Dots used for a particular color, where N specifies a color from 0 to 5. Value valid after a write access to *DotCountSnap*
0x30	DotCountSnap	1	0x0	Write access causes the

528

				AccumDotCount values to be transferred to the *DotCount* registers. The *AccumDotCount* are reset afterwards.(Reads as zero)
0x34	PhiHeadSwap	1	0x0	Controls which signals are connected to printhead channels A and B 0 - Normal, specifies bit 0 is channel A, bit 1 is channel B 1 - Swapped, specifies bit 0 is channel B, bit 1 is channel A.
0x38	PhiMode	1	0x0	Indicates whether the PHI is operating in master or slave mode 0 - Slave Mode 1 - Master Mode
0x3C-0x40	PhiSerialOrder	2x1	0x0	Specifies the serialization order of dots before transfer to the printhead. Bus 0 - Printhead Channel A Bus 1 - Printhead Channel B If set to zero the order is *dot[1:0]*, then *dot[3:2]* then *dot[5:4]*. If set to one then the order is *dot[5:4]*, *dot[3:2]*, *dot[1:0]*.
0x44-0x48	PrintHeadSize	2x16	0x0000	Specifies the number of non-margin dots in the printhead ICs (must be even). If margining is to be used then the configured *PrintHeadSize* should be adjusted by the dot margin value i.e. *PrintHeadSize* = (*Physical-PrintHeadSize* - (*DotMargin* * 2)). Value must be divisible by 2 (i.e. the low bit must be 0) Bus 0 - Specifies printhead on Channel A Bus 1 - Specifies printhead on Channel B
CPU Direct PHI Control (See Table 213.)				
0x4C	PrintHeadCpuIn	3	0x0	PHI interface pins input status. Only active in direct CPU mode (Read Only Register)
0x50	PrintHeadCpuD	3	0x0	PHI interface pins direction control.

	ir			Only active in direct CPU mode
0x54	PrintHeadCpu Out	10	0x000	PHI interface pins output control. Only active in direct CPU mode
0x58	PrintHeadCpuC trl	1	0x1	Control direct access CPU access to the PHI pins 0 - Normal Mode 1 - Direct CPU Control mode
0x5C	Print-HeadCpuCtrlM ode	1	0x0	Specifies if the pin is controlled by the *PrintHeadCpuOut* register or by the Fire generator logic. Only active when *PrintHeadCpuCtrl* is 1 and pin is in output mode. Bit 0 - controls the *frclk* pin When the bit is 0 - Pin is controlled by *PrintHeadCpuOut* 1 - Pin is controlled by Fire Generator Logic
Line Sync Control				
0x60	LsyncHigh	24	0x00_0 000	In Master mode specifies the number of *pclk* cycles *phi_lsyncl* should remain high. In Slave mode specifies the minimum number of *pclk* cycles between *Lsync* pulses. *Lsync* pulses of a shorter period will cause the PHI to halt due to buffer underrun.
0x64	LsyncLow	16	0x0000	Number of *pclk* cycles *phi_lsyncl* should remain low.
0x68	LsyncPre	16	0x0000	Number of *pclk* cycles between *PrintStart* rising transition and the generated *phi_lsyncl* falling edge
0x6C	LsyncDeglitchC nt	4	0x3	Number of *pclk* cycles to filter the incoming *Lsync* pulse from the master. Only used in slave mode.
0x70	LineSyncInterru pt	16	0x0000	Number of line syncs to occur before generating an interrupt. When set to zero interrupt is disabled.
Shift Register Control				